1988

Smalltalk and robotics :

Torez Hiley
Lehigh University

Smalltalk and Robotics

A Simulated Approach To Programming Robots

by

Torez Hiley

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for Degree of

Master of Science

in

Computer Science and Electical Engineering

Lehigh University
1987

This Thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

December 15, 1987
(date)

Lawrence J. Vanneman
Chairman of Department

Donald J. Hillman
Professor in Charge

ACKNOWLEDGMENTS

I would like to take this time to acknowledge all those who have helped to bring about this paper and the various concepts that are expressed.

I would like to thank Dr. Roger Nagel for the initial inspiration and interest in the study of robotics; Dr. Donald Hillman for providing counseling throughout my research period on this document. I would like to thank Scott Hawker for guidance and support in the direction of this project. Special thanks to Wilma Davidson and Renel Smith for endless readings and critiques of earlier versions of this document.

CONTENTS

LIST OF FIGURES

Smalltalk and Robotics:A Simulated Approach to Programming Robots

Torez Hiley

Lehigh University

ABSTRACT

A useful robot must be autonomous and free of external control from a human operator. It must be a general purpose device that, on different occasions, can perform different functions. It must interact with its environment and have at least a limited self-awareness, act on this awareness and receive feedback describing the result of the action. To do all these things requires that a robot be under the control of a programmable computer and have senses [1].

To date several programming languages exist to meet the needs of the programmable computer. Smalltalk-80 is yet another language. This document analyzes the Smalltalk-80 environment by comparing and contrasting it with present robot languages. The characteristics and qualities of an ideal language are introduced. Smalltalk-80 is then analyzed a little closer to see just how it fits into the model of the ideal language. The advantages and disadvantages of the environment are weighed, taking object oriented programming into account. Finally, based on a simulated program, the document concludes Smalltalk-80 is a powerful language for robot programming.

1. Chapter 1: Introduction

Efficient communication with industrial robots is a key factor in the success of contemporary programmable automation, but robot languages--the means of communication--have commonly been developed in an ad hoc manner to meet the needs of a particular robot and application. Thus, we currently have almost as many robot languages as robots. This chapter looks at several of these languages and the categories they fit into. Because this field is so young, this chapter draws heavily on the limited resources available.

Chapter 2 offers a quick look at the Smalltalk-80 environment, defining object oriented programming, classes, sub-classes and the like. It concludes with an analysis of advantages and disadvantages to the object oriented approach and why such an approach for robotics. Chapter 3 breaks down the specific features of the Smalltalk-80 environment, keeping in mind the possibility of using it for robot programming.

Chapter 4 ties chapters 1, 2 and 3 together. It takes another look at the levels of programming and applies them to specific features of Smalltalk-80. It makes use of sample code to demonstrate the findings. Chapter 5 makes use of a full and complete simulated program. It defines the motivation and layout of the program, thus concluding the environment can be used to simulate robot programs.

Chapter 6 concludes the document by summarizing the findings and suggests areas for future research. Though the document emphasizes simulating robot programs, chapter 6 discusses the steps involved in making that simulated program a reality.

Thus, actually programming the robot can be interspersed with simulated instructions and down-loaded to the robot through the output ports.

1.1 Basic Requirements

Most of the existing languages have historically been designed for standalone systems, concentrating on manipulator control and tending to ignore data manipulation. They can be divided into four loosely formulated levels according to language emphasis from simplistic point-to-point to complex task-oriented problem solving. Overlaps between levels occur but do not interfere with the basic features of each[2]

Regardless at which level the language is categorized, it must contain the following basic requirements:

1. Portability.
2. Well documented software (clarity of software).
3. English-like commands so that they are easy to understand (simplicity of software).
4. Commands that would apply to any robot or peripheral device in the environment (portability of software).
5. Unity among the commands. All are usable and not just some.
6. Clarity of program structure, for example, looping structure, should be easy to follow through.
7. Naturalness for the application. Although the structured languages may not have builtin robot commands, they allow subroutine defining in which the user can define a suitable command for the robot.
8. Expandable software.
9. Manual teach mode for robotic applications.
10. Trace features for debugging purposes.
11. Halt execution capabilities for debugging purposes.

12. Back stepping the robot for debugging purposes.
13. Break points are easily incorporated.
14. Immediate execution (property of interpreter languages).
15. Efficient language.
16. Decision making capabilities can be implemented in the language.
17. Interaction with world modeling systems.
18. Learning, self repair and self organization.
19. Collision detection and avoidance.
20. Error handling and recovery.
21. Development independent of robot.
22. Off-line debugging and development facilities that use computer graphics techniques to simulate the robot and its environment.
23. Arm synchronization and parallel processing.
24. Use of sensory data/feedback.
25. Representation of transformation and frames.

1.2 Four Levels of Programming

1.2.1 Point-to-Point Programming Point-to-point languages provide programmed robot control. They enable the user to save a series of points obtained by guiding the robot through the required motions. Usually the guiding is done by using a manual device that activates the joint motors and saves desired locations; or while moving the robot itself and saving points continuously. Higher level guiding systems provide specialized function buttons which allow editing of programs and interaction with external signals.

Advantages

- Once a task has been programmed in a point-to-point language, it can be repeated any number of times without operator intervention.
- Programs are easy to debug because verification is constantly being done on the robot itself.

— Some more advanced point-to-point languages will provide the user with the simple branching and subroutine capabilities, more powerful sensing, primitive parallel execution, and some attempts at frame definitions.

Disadvantages

— Emphasis is placed on robot motion rather than on robot task.

— There exists no software to handle emergency situations and no expandability of off-line programming.

Example 1

Some languages, such as those provided with the Anomatic II Controller, provide a powerful numerical control language with programmable mathematical expressions, variables, jumps and subroutines, and self-configuring capability. It is an interpreter language that allows 10 levels of routine nesting. It provides simple and unconditional branching. Motion can be specified by using the joint angles directly or in Cartesian coordinates. Files can be included as executable code by running the files as subroutines.

Example 2

Funky [2], on the other hand, is an advanced guiding system that produces robot programs through the use of manual guiding and a function keyboard. It has a joy stick to control the motion of the robot. Control of the system is similar to a cassette tape recorder. The *play, erase, record, reverse,* and *fast forward* modes are all comparable to their cassette player counterparts. These modes allow insertion and deletion of points and stepping forward and backward through the program. It provides a command to center the gripper about an object, using touch sensors in the hand, and another command to operate an electric screwdriver on an additional gripper. It is a compiled,

assembled and interpreted language in one.

Example 3

The MIC [3] language is compiler based and was designed to control the TeachMover robot. It is a first generation control language with extended features such as:

— The programs are transportable from one TeachMover robot to another, in either source or binary format.

— Provides additional robot control commands which are not available through the teachbox (like the PATH command).

— Compared to the teachbox method, program editing is easier because the user can insert program steps and not just overwrite them.

— All robot movement calculations are done "off line" before they are actually needed. This increases the speed of the compiler.

— Can communicate with external devices.

Under the MIC system, the user has the ability to check for objects in the gripper as well as the object's size. It provides absolute as well as straight line motion. Allows simple conditional branching and a way to vary the speed of the robot. Must have a Pascal compatible host computer with an asynchronous serial communications port in order to run the compiler.

Example 4

Sigla [2] provides features such as parallel task control and variable instruction sets for software tailoring. It is an interpreted language and provides simple and unconditional branching. Provides program nesting and subroutine calling. Files can be included as executable code by running all files as subroutines. Provides absolute motion that can be specified by using

the joint angles directly or in Cartesian coordinates.

— Simple parallel processing, in the form of mutually exclusive operation of arms with limits and convergence points to ensure collision avoidance.

— This is done by allowing execution of several different files on several different arms all at the same time and it provides an anticollision command that sets up work boundaries for the different arms.

1.2.2 Structured Programming Structured robot programming languages incorporate structured control constructs and permit the extensive use of coordinate transformations and frames. They include complex data structures, improvements in sensors and parallel processing, the use of predefined state variables, the use of predefined subroutines and parameter passing. Sensor commands are similar to those on the point-to-point level, but wide variation in sensing capability does occur among the languages. The languages have semaphore primitives activated only when a given event occurs. Motion is defined in terms of the transformation on the frame of the robot hand.

Program understandability is greatly increased and aids task-oriented programming by the introduction of more sophisticated parallel processing techniques and state variable concepts. At this level, point-to-point programming is replaced by the manipulation of object frames. Because of the complex data and control structures, the languages themselves are inherently more powerful than those at the previous level. Off-line programming is more feasible as long as relational transformations are accurate, and any discrepancy between the model and the robot environment

can be expressed in terms of a transform.

Advantages

— Structured programming languages can interact with external devices.

— Parallel processing has been expanded from the point-to-point level.

— The programs are expandable.

Disadvantages

— Structure programming techniques are difficult to use and require more user education than do point-to-point languages.

— Although more advanced than the point-to-point languages, the structured programming languages are also less practicable for today's robot applications, hence, the noncommercial origin of languages in this level.

Example 5

AML/X [4] is an *expression-oriented* language, as opposed to a *statement-oriented* language. Thus, expression evaluation is the fundamental computational process in AML/X. It is a structured language which allows recursion and subroutine definition. It is an interpreted language that allows data abstraction.

The basic data grouping mechanism for AML/X is the aggregate; however, it supports a large number of data types and allows the user to define new ones. Many operators and builtin subroutines can handle several different data types. The language allows the user to raise exceptions as well as define exception handlers. It helps recover from unexpected situations. Allows a connection to be made between the operating system and AML/X via channels. Allows communication with external devices.

Three points about the language can be categorized as disadvantages; it does

not support multitasking; does not use standard syntax; the language is slow.

Example 6

LISP [5] is an interpreted language that provides the user with the ability to extract and combine elements and lists from lists. Instead of a program described as a series of steps, LISP forms complex functions from other simpler functions. A function inputs one or more lists and outputs one or more lists.

Lisp is used with laboratory robots, especially when scientists wish to write programs that cause the robots to exhibit artificial intelligence.

All of the intermediate storage used to obtain the final result is reallocated by a "garbage collector." Lisp is good at manipulating symbols-of ideas, concepts, or physical objects like "dog" or "flower."

Because recursion is the intimate part of the programming style in LISP, it is confusing to programmers' use to higher-level languages (structured languages) without so much recursion.

Example 7

Prolog [6] is an interpreted, conversational language designed to make a computer simulate the thinking process by making deductions from information given in logical formulas. It is used for solving problems that involve objects and the relationships between objects. It is a declarative language, in which the programmer declares known facts using predicate

logic.

It can go from a goal state back to a series of previous possible conditions (backward chaining).

The programmer can enter facts and rules directly into Prolog and then ask questions about the facts.

It allows the representation of frames through structures.

Example 8

QLISP [7] is both a programming language and an interactive programming environment. It grew out of the QA4 language that was developed at the Stanford Research Institute from 1969 through 1972. QLISP embeds an extended version of LISP with a variety of sophisticated programming aids.

QLISP provides a rich set of data types and facilities for manipulating them. Expressions composed of any data types may be placed in a data base, where the data can be fetched by content rather than by name or address. By storing all data in a common discrimination net, QLISP can represent equivalent expressions uniquely. Arbitrary expressions are represented uniquely.

In addition to the range of types including numbers, arrays, string, list and binary tree structures, QLISP provides data of type TUPLE, VECTOR, BAG and CLASS. It provides a unification pattern matcher in which each of two expressions may act as templates for the other.

1.2.3 State Table Programming Each module in a hierarchical control system can be represented by a finite-state automaton [8]. At each level, input commands from the next higher level are decomposed into sequences of output sub-commands to the next lower level in the context of the state of the environment, of the state of the control system, and the internal store of knowledge. At each level predictions and expectations are generated by the internal world model in the context of the state of the task, the goal of the system, and the best current hypothesis about the state of the environment. Also at each level, processed signals from the environment are compared against expectations from the world model.

Advantages

- The task can be partitioned into simple, well defined modules with clearly specified inputs, outputs, internal states, and rules for state-transitions.
- Design becomes simple and synchronization of simultaneous processes becomes easier.
- It allows expandability with little or no effort.
- Debugging is simple because each control state in the system is formally identified and the set of conditions that lead to and from that state are clearly specified.
- It is possible to build teaching and learning capabilities into sensory-interactive robot control systems.

Disadvantages

- Each possible input and output for a given state must be represented. This can give rise to a lengthy program whose function may not be that advanced.
- The system is slow.

1.2.4 Task Oriented Programming At this level of robotic programming, an operator would be able to direct a robot as he would direct a person, to perform a series of tasks, which the software would then determine how to carry out [9]. The

use of a world model and concealing low-level aids from the user (like sensors and coordinates) are also key factors.

Advantages

— Controlling the robot becomes simple because the commands are English like commands.

— In the place of the low-level explicit languages that tell the robot what to do in great detail, such as "*raise joint #4 nine millimeters at an average velocity of 20 mm/second*", task oriented languages use the high-level implicit commands, such as "*load pallet*", or "*weld fixture*".

— These languages make use of the world model which allows for a detailed description of the environment as well as accepting and issuing feedback.

Disadvantages

— The high level of the commands necessitates the use of a complex world modeling system, artificial intelligence for decision making, and an interactive debugging system.

Example 9

Autopass [2] is a high-level programming system for computer-controlled mechanical assembly. It is oriented towards objects and assembly operations that enable the user to concentrate on the overall assembly sequence and to program with English-like statements using familiar names and terminology. It is an interpreted type language and is designed to resemble the assembly instructions humans might use.

Autopass uses high-level commands such as *place* object1 *on* object2. The execution of this command involves finding and identifying object1 and object2, determining a pickup point and vector for object1, moving to pick up object1, deciding where on object2 to place object1, placing object1 on object2, and remembering the new relationship between them. To keep track

of objects, Autopass absolutely requires a world modeling system and must also be capable of making assembly-oriented decisions, such as how to pick up an object. The ideal world modeling system would involve vision location and identification and tactile sensors for help in locating and picking up objects.

Autopass has implicit continuous path motion. Its statements mean precisely what you think they mean and are therefore easy to understand and use. Commands are interpreted into lower level code, whose validity the user must verify. They can alter any segment of the code, and the compiler can question the user about any ambiguities.

In spite of its advancements, Autopass does not solve the problems of collision avoidance and emergency decision making. Its high level statements lead to ambiguities between the user's intended actions and the robot's interpretation of them.

Example 10

RCCL [10] is a manipulator language written in the **C** computer programming language running under the UNIX® time-sharing system. World representation and motion primitives necessary to describe a manipulator task are provided by a set of system calls and predefined data-structures.

It is a task oriented language that supports straight line as well as joint interpolation motion. It supports a transform, referred to as *hold* transform, which can be asynchronously updated by the user process and their values will be taken into account only when the corresponding motion begins. The

values can be accessed or modified during the execution of the task. Force control is used when a manipulator is to come into contact with rigid objects of the environment such as when mating parts.

RCCL is an addition to the C language of system calls. It therefore benefits from the advantages of the UNIX® system as far as maintenance, portability, macro definition, standardization and modularity are concerned. It is easy to interface with other languages supported by UNIX® and it provides input-output operations as well as user interaction. It allows for Cartesian path generation, mathematically described trajectories, and conveyor tracking (e.g. moving coordinate frames).

One drawback with RCCL is that external sensors are not distinguished from internal sensors. This causes complications when the robot is responding to its environment.

1.3 Tomorrow's Dream

The robot of the future possess a high amount of intelligence. It is indistinguishable from humans in physical and mental development and it remains so throughout a life cycle identical to humans. Considering the robot does not initially inherit this intelligence, it must be programmed. To program the robot requires that one has a good definition of intelligence.

An entity is intelligent if it has an adequate model of the world, if it is clever enough to answer a wide variety of questions on the basis of this model, if it can get additional information from the external world when required, and can perform

such tasks in the external world as its goals demand and its physical abilities permit. Hence, an ideal robot language would not only encompass the basic requirements but would:

— provide the robot with the capabilities to answer questions on the basis of this model;

— provide the robot with an understanding of its own goals;

— have the ability to interact with the external world;

As this chapter concludes, there does not exist an "ideal robot language" in the current generation of languages. In comparing the point-to-point languages with the structured and the task oriented languages, we can see the progress made thus far. However, can the open issues, like software portability, adequate world model, learning, etc, be solved ? Can we, in fact, give birth to that long awaited language ? Or will it always remain a dream ?

2. Chapter 2: Smalltalk-80 At A Glance

2.1 What Is It?

Smalltalk-80 was released for general licensing in May, 1983, by the Xerox Corporation. [11] It consists of an object-oriented programming language and an integrated collection of tools for interacting with components of that language. In the Smalltalk-80 language, the fundamental way to indicate that something should happen is by sending a *message* to an *object*. The *object* is a representation of information consisting of private memory, and a set of operations to manipulate information stored in the private memory or to carry out some actions relative to that information. Sending a *message* is the Smalltalk-80 way of asking the *object* to carry out one of its operations.

All information in the Smalltalk-80 language is represented as *objects*, each one knowing the *message* it can understand. Associated with each *message* is a *method* that describes how the *object* should respond to the *message*. For example, the user interface to the Smalltalk-80 system can be viewed as a graphical way to identify *objects* and to choose *messages* to send to *objects*. When an *object* is sent a *message*, the appropriate *method* is invoked and some action is taken.

The Smalltalk-80 environment is based on windows and views, accessed and manipulated via an attached keyboard and pointing device. Although it is defined on top of a Unix-like kernel, it is a menu driven system which includes such utilities as compiler, debugger and text editor. Its graphics make reality a little more visible while its debugging mechanism is a programmer's dream. The

language syntax is far from desirable but it tends to get its *message* across.

2.2 Objects, Messages, Methods and Classes

The Smalltalk-80 language is based on a uniform use of objects and messages. An *object* represents a component of the language. For example, objects represent

- numbers
- character strings
- queues
- dictionaries
- rectangles
- file directories
- text editors
- programs
- compilers
- computational processes
- financial histories
- views of information

Objects are a uniform representation of information that is an abstraction of the capabilities of a computer. The two capabilities of interest are the capability to store information and the capability to manipulate information. Thus, an object has private memory and a set of operations.

The nature of its operations depends on the type of component it represents. Objects representing numbers compute arithmetic functions. Objects representing data structures store and retrieve information. Objects representing positions and areas answer inquiries about their relation to other positions and areas. An object carries out one of its operations when another object sends it a message to do so.

Each object knows the messages it can understand.

A *message* is a request for an object to carry out one of its operations. A message specifies which operation is desired, but not how that operation should be carried out. The *receiver*, the object to which the message was sent, determines how that message is to be carried out.

The set of messages to which the object can respond is called its *interface* with the rest of the system. The only way to interact with an object is through its interface. The private memory of an object can only be manipulated by its own operations. The only way to invoke an object's operations is through its messages. These properties promote modularity. They insure that the implementation of one object cannot depend on the internal details of other objects, only on the messages to which they can respond.

The concept of *method* is analogous to *procedure* and *function* calls of other programming languages. It is associated with each message and is a procedure that describes how the object should answer the messages sent to it.

Linking these concepts to ones more familiar, an object is like a computer consisting of data and procedures that operate on that data. An operation is invoked by calling on some procedure or method. Sending a message is the Smalltalk way to invoke procedures. Smalltalk is a simulation of many computers communicating with one another.

There are many objects in the Smalltalk-80 system. Objects that respond to the same messages in the same way are grouped together. When they are grouped

together, their private memory is represented in the same way and their methods refer to their data with the same set of names. A group of objects related in the same way is called a *class*. Objects in a group are called *instances* of the *class*. Programming in the Smalltalk-80 language consists of creating new classes, creating instances of classes, and specifying a sequence of message exchanges among all of these objects.

Every object in the Smalltalk-80 system is an instance of a class. All instances of one particular class represent the same kind of system component. Classes have names that describe the kind of component their instances represent. Instances of a class named `Point` represent spatial locations. Instances of a class named `Rectangle` represent rectangular areas. Instances of a class named `Process` represent independent processes.

Refining existing class descriptions is a powerful way in which to approach Smalltalk-80 programming. Such refinement is supported by the ability to create a subclass of an existing class. A subclass describes a group of objects that inherit information from an already existing description. A subclass can add new functionality or private memory, and modify or prohibit existing functionality.

2.3 Advantages of Object-Oriented Programming

Programming in the Smalltalk-80 language consists of identifying objects, classifying them according to similarities and differences, and designing a language of interaction among these objects. These are important organizing skills and communication skills that can be taught using this form of computer programming.

What are the advantages to this form of programming?

1. The information known privately to an object is protected. This information can only be accessed directly by the methods of the object. This means that the structure of an object (the representation of the information of an object) can be changed without affecting interactions with instances of other classes. This ensures that there is a structure or discipline by which objects interact and that a user can make changes or additions to very complex systems without getting caught in a maze of interdependencies.

2. The user accesses existing objects as well as creates new ones or modifies existing ones. Modification is done by adding a new message and its method to a class description, or by adding new data slots to private memory of all objects in a class. This means that the Smalltalk-80 language provides a simple and expressive model for the relationship among parts and wholes so that the process of building a system can draw on one's intuitive ability to synthesize and analyze.

3. The Smalltalk-80 system is built on the model of communicating objects. Large applications are viewed in the same way as the fundamental units from which the system is built. The interaction between the most primitive objects is viewed in the same way as the higher-level interaction between the computer and the user.

4. Objects support modularity. The complexity of the system is reduced by minimization of interdependencies of system parts. Complexity is further

reduced by grouping together similar parts, where this grouping is achieved through classes. Classes are also the chief mechanism for extension in the system. And subclasses support the ability to factor the system in order to avoid repetitions of the same concepts in many different places. Managing complexity is a key contribution of the Smalltalk-80 approach to software.

2.4 Why Smalltalk-80 and Robotics?

A robot program application must provide many views[1] of its subject. The primary view of the action is static, the arm and the object it is carrying is stable. The primary view of the model is dynamic. The entire plane or surroundings are constantly changing. The programmer's insight into the program's behavior comes from merging these views: examining data within the process while controlling its progress through the task. Debugging purely at the task level may be sufficient for a safe language, correctly implemented, but in practice we also need views of the implementation[12].

A particular view is not used in isolation; it is related to other views. The programmer moves rapidly among a set of related views. This diversity complicates the user interface to the program. Any attempt to base an interface on a coherent model is confounded by the user's need to switch among so many projections of the underlying subject. For example, the keyboard input

1. Smalltalk-80 defines a view as a rectangular area on the display screen in which to access information.

could mean "pick up the object at location 100" of the current view. It could also mean "move the arm to location 100" or "place the object at location 100" and so on. There is no natural interpretation; it depends on the programmer's current focus.

Object oriented programming is better suited to the implementation of robot programming than sequential or multiple sequential process programming. Under sequential programming, a process executes a program. At any time the process is in some state, with control at some location in the program. When the process blocks for input from the user, it is in the state from which it resumes when the user responds. This makes it difficult to drive the process with flexibility, where the user is allowed to refuse a given menu and leap to some unrelated context. The process must accept either the menu selection or the context switch. The menu selection is a local operation in the local context; the context switch is a global operation involving unrelated parts of the program. This might be achieved by letting the user traverse a network of contexts. But there is then a tradeoff between ease of use and ease of programming. A tree is easy to program, but tedious for the user to traverse. A fully connected network might be easy to use but calls for every part of the program to be intimately coupled to every other part.

With multiple sequential processes a separate process could be associated with each context. As the user moves around the graphics screen, input is directed to the appropriate process by some agent that maps screen locations to processes. Processes are suspended until the user needs them. They need only know about the

semantically related process. As a model this yields a natural architecture, but its implementation is usually very expensive. The overhead for creation and interaction of separate processes might be acceptable for a small number of processes, say one per window. But the desired user interface has each line within each interacting independently with the user. Many hundreds, even thousands of processes would be needed. With conventional multi-process techniques, the cost is prohibitive.

The object-oriented approach lies between these two approaches. Instead of a collection of processes, there is a single process containing a collection of *objects*, each an instance of a *class*. Each *object* has a copy of the *data members* defined in its class and executes the *function members* of the class. Unlike a process, an *object* has no permanent state other than its data. *Objects* share a universal address space and communicate with one another by invoking function members as procedures. It is feasible to have many thousands of *objects*. That *objects* cannot execute concurrently is not significant. Such concurrency is not really required.

3. Chapter 3: A Closer Look

3.1 The Environment

Smalltalk-80 is a graphical, interactive programming environment. It is designed so that every component in the system that is accessible to the user can be presented in a meaningful way for observation and manipulation. The components include objects that provide the functions usually attributed to a computer operating system: automatic storage management, a file system, display handling, text and picture editing, keyboard and pointing device input, a debugger, a performance spy, processor scheduling, compilation and de-compilation.

The Smalltalk-80 system supports a number of interesting design tools, notably classes and instances as units for organizing and sharing information, and subclassing as a means to inherit and refine existing capabilities. Combined with the interactive way in which the program development process is carried out, the Smalltalk-80 system provides a rich environment for prototyping new applications and refinning existing ones.

All information that you can seek about the system involves information about existing objects or about objects that can be created. You can find out information about the internal state of an object using the view called an *Inspector*. You can find out about the message interface to an object using a system view called a *Browser*. A *System Browser* gives you access to all the class descriptions available in the system, including comments about the classes, comments about the methods, and examples of how to use many of the classes. Other ways to find out about

messages and methods involve creating system views called *Message-Set Browser.* These views are created in response to queries to determine which methods send a particular message, which classes implement a particular message, or which methods reference a particular variable or literal.

Comments. Each class description includes a comment about the purpose of the class. A class comment is obtained (for reading or for editing) by choosing the yellow button command `command` in the class names subview of the browser. Other comments document the purpose of a method. These are found by choosing a message selector; the comment is the text within double quotes at the beginning of the method definition. Programmers can also document the design of a method by interspersing quoted text within the method itself.

Explanations. Suppose you want to understand an existing method. One form of explanation could be explanation about the tokens that appear in the method. You can select any token and then choose a command to obtain a short description of the role of the token.

Because of the class and message structure of the system, it is possible to provide explanations about which messages are sent in a particular method, and to obtain browsers to answer queries as to which methods send a particular message, which classes implement a particular message, and which messages are sent in a particular method. These are the kinds of queries that a programmer must be able to make to determine the structure of the

Smalltalk-80 system. The system browser as the program editing interface provides the framework for both the incremental development of class descriptions, as well as the context for accessing information about classes. Each new class added to the system by the programmer is accessible via a browser and can be queried in the same way as all system-provided classes.

Templates. Whenever the system "knows" something about the form in which information should be provided, a template or a default solution is provided. The user edits the template, replacing descriptive words with the actual desired text. This means that the user does not have to remember syntax and can be prompted on the kind of information that is required. Templates are used to assist in defining classes and methods, and in commenting the system.

Examples. In many of the descriptions of classes in the system, there are messages that consist of examples of how to interact with instances of the class. Examples show how to create a new instance or how to use an instance. These examples are typically messages to the class itself, where the method includes documentation comments and expressions to be evaluated.

Menus. Menus are a form of assistance in the system. The items in a menu denote and remind you of the kinds of activities you can do. You choose an activity by pointing to the menu item, rather than having to remember the correct key words and having to type the words correctly. Items in a menu represent the behavior of an object; most likely, there is a message in the class description of the object that carries out the behavior represented by the menu item. You

can learn about programming in the system by searching for these menu item/message correspondences.

3.2 Spelling Errors

A major area of research in the development of programming environments has been to provide on-line assistance to the programmer. Some of the earlist work in this area was done in the context of the development of the Interlisp programming environment under the research title of "Programmer's Assistance."[13] [14]

One of the results of this research was the introduction of the DWIM, or "do-what-I-mean" spelling correction, approach to interaction and error handling.

Adapted from this research on spelling correction, the Smalltalk-80 system includes the ability to assist the programmer by correcting misspellings of variable names and message selectors. Spelling correction is done within the context in which you are working, that is, within the scope of the variable and message selectors of a class definition or of an interrupted context.

3.3 Debugging

There are numerous ways to debug programs or a system of programs. Smalltalk-80 *methods* print trace messages by sending them to the System Transcript. The System Transcript typically appears at the top left corner of an initial Smalltalk-80 display. It is a standard system view, so it can be moved, framed, collapsed and closed. When a class is compiling or in the process of being written to an external file, the status appears in the Transcript window. When users are debugging

methods and it is desirable to print periodic messages indicating the state of evaluation, the System Transcript is a useful place in which to display the messages.

Embedded trace statements prove to be quite useful but they fail to provide varying details. Smalltalk-80 support a debugging system to get around that. A notifyer is used to provide a simple description of an activity or *process* at a time the activity is interrupted. Interruption can occur because of a run time execution error, or because the user purposefully cause such an interruption. This purposeful interruption is accomplished by typing *control-c* on the keyboard. The notifyer displays the reason for the interruption in the label of the view. The text in the notifyer indicate the last few messages sent before the interrupt occurred. The methods associated with these messages have not yet returned their values. For example, if a message sent to a method was not understood by the method, system execution halts and the notifyer of Figure 1 appears.

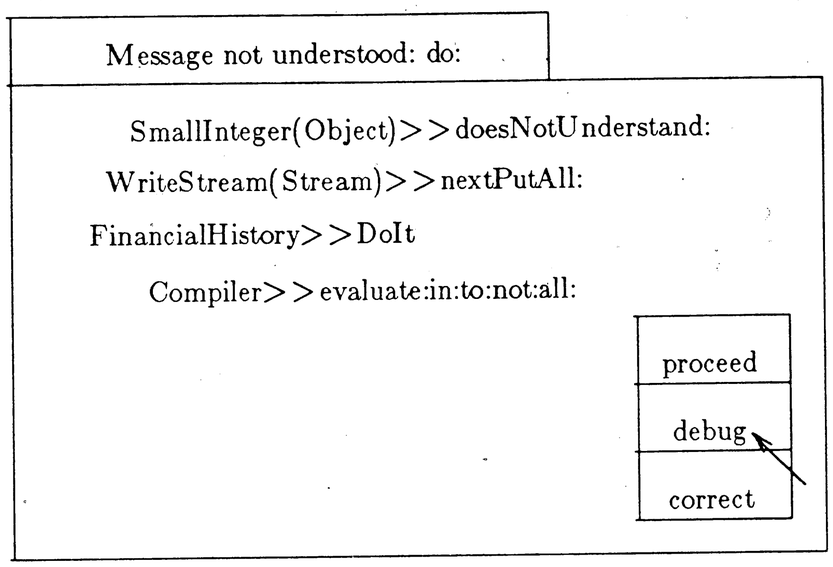


Figure 1. Invalid Message Sent To A Method

The notifier is labeled **Message not understood: do:**. The text displayed in the notifier indicates the last few messages sent before the interrupt occurred. The methods associated with these messages have not as yet returned their values. This sequence represents the *activation stack* that the user wishes to explore to understand the source of the error. The user is given the choice to continue program execution, correct the error or further debug it. Selecting **debug** creates the debugger view as shown in Figure 2.

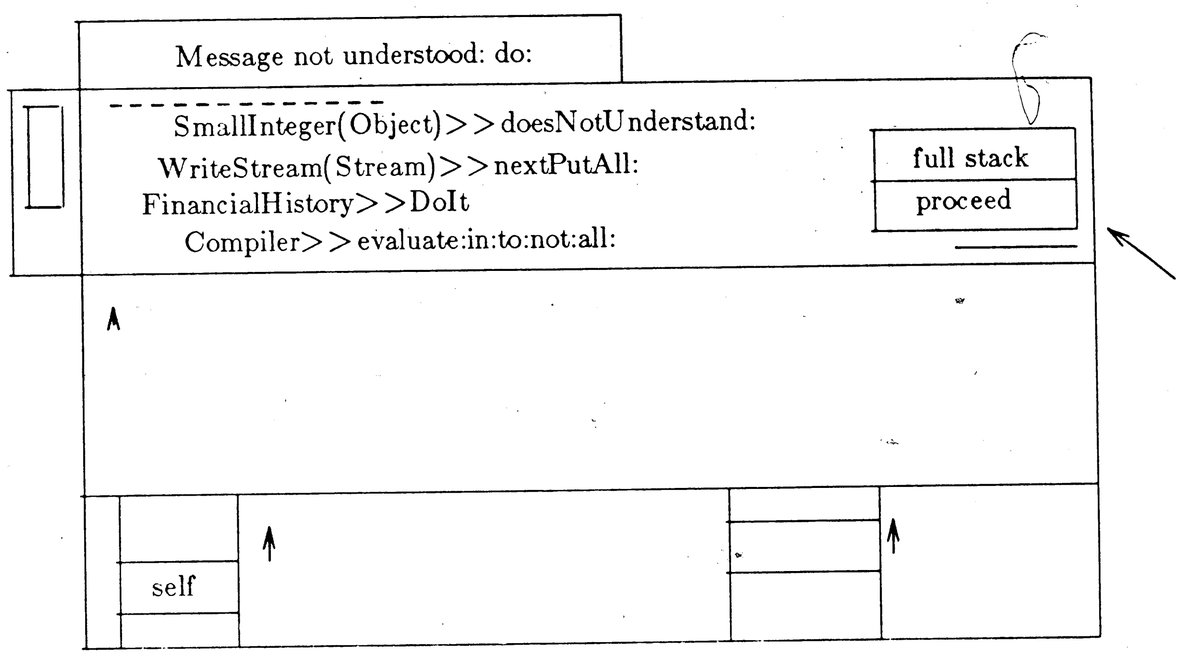


Figure 2. Debugger Subview

A debugger view is made up of six subviews. The top subview is a menu of classes and messages on the *activation stack*. The second view is a text view in which the method associated with a selected message is displayed. The items in the class/message menu are the message-sends of the interrupted execution, identical in format to the message-sends displayed in the notifier. Each message-send displays the class of the receiver and the message selector of the message sent to the receiver. It also shows the class in which the interpreter found the method for this message selector.

The debugger presents some or all of the sequences of message-sends that occurred. It allows the user to select each one in order to see the method and to determine at which point in the method the interrupt occurred. The user can choose any message-send on the stack and cause evaluation to proceed from this selected point.

They can single-step through message-sends, checking the state of the variables in order to determine the source of the error. They can change the value of variables and proceed. If they want to evaluate expressions within the debugger view, evaluation is carried out in the context of the currently selected message. The user can also edit and recompile a method.

3.4 Graphics

3.4.1 Display Taking all possibilities into account during the motion of an item is not an easy task for a programmer. Kinematics and the laws of physics play important parts in displaying the items. If for example the robot arm dropped a ball it was carrying, the speed of the falling ball is depends on the gravity forces afflicted on it. As it hits the surface beneath, it bounces to a height that decreases with each bounce. What if the ball fell on top of an egg causing it to crack open? You now have three objects moving simultaneously; the arm reacting to losing the ball, the ball decreasing in velocity and height as it bounces, and the egg splattering even more as the ball bounces on top of it.

Defining each item in the display as a Smalltalk object takes much of the responsibility away from the main interface controlling them. Each item is then responsible for updating itself and passing it is new form and coordinates to the interface for display. Modularity and portability is a gain in this approach. It removes dependency from the controlling interface knowing the properties of each item displayed. And, any possible items that are later added. What is required is that the items respond to the same system dependent message, in the appropriate

manner.

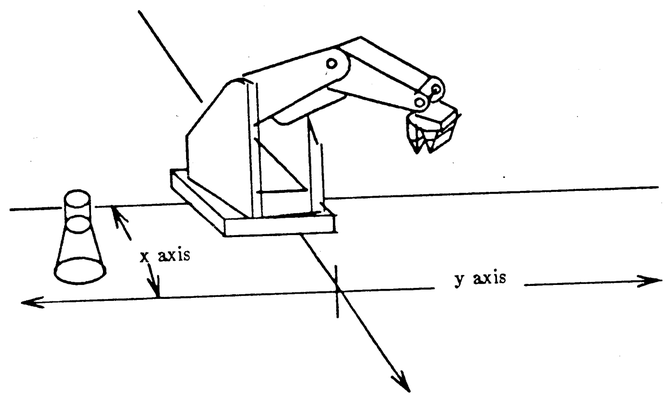


Figure 3. Before Collision

3.4.2 Collision Displaying an item can be as simple as changing its rotation on the x-y axis or as complex as an egg cracking into many small pieces. For example, consider the display as shown in Figure 3. Suppose the arm rotates on the y-axis in preparation to pick up the jug. Without a collision avoidance algorithm, the rotating arm tips over the jug it was going after, as shown in Figure 4. Displaying this motion is as simple as changing the frame of the items. That is, when collision is detected, each item involved is notified. The object's reaction changes the view of the item. In this case, the pictorial view remains the same but its location and orientation on the axis has changed.

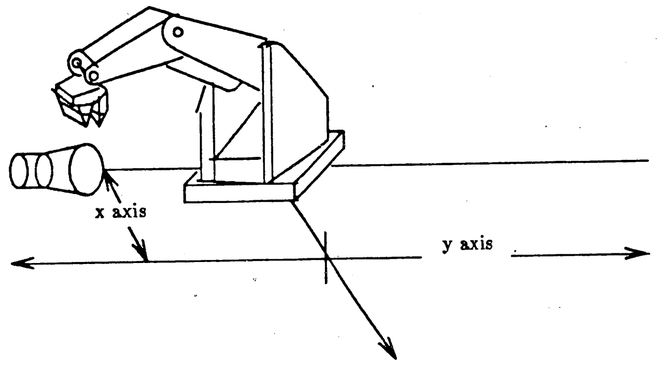


Figure 4. After Collision

An algorithm to detect and avoid collision is easily implemented in Smalltalk-80. Before an item makes a move, it searches the path it is traveling for another item. If one is in the way, a reaction occurs. In the case of a moving arm, motion can be halted and the system notified that there is something in the way of the arm that could possibly cause a collision. In the case of a ball falling on top of another ball, for example, collision cannot be avoided. It does not make sense to stop the falling ball in thin air and say, "Hey...change your course because you will hit something if you continue to fall." Instead, the ball continues to fall. When collision occurs, both balls begin to bounce until they die down.

4. Chapter 4: The Four Levels Incorporated in Smalltalk-80

After looking at existing robot languages, this chapter maps those features into the Smalltalk-80 Environment, thus stating how the environment incorporates the same features. It uses sample code to demonstrate a way to program the environment to implement the 4 levels of programming.

4.1 Point-to-Point

Point to point can be implemented in a way that allows the user to literally "show" the system where the robot is to be moved. This motion can be displayed in a window as a series of dots (see Figure 5). Each dot will represent a point in the path which the robot is to take. The current point is so marked by placing an **X** over its dot.

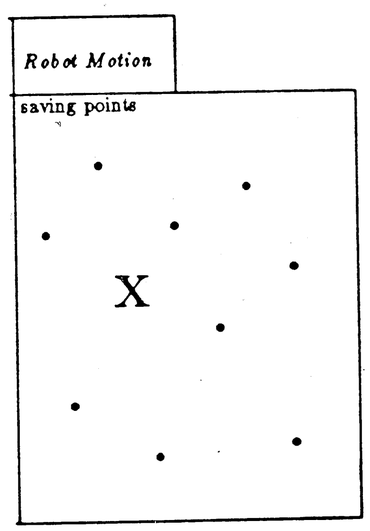


Figure 5. Motion Displayed As Points In A View

To aid in defining and debugging the path the robot is to take, the mouse buttons are programmed to perform special functions. The red button is used to add the

points/dots to the path/window. The yellow button answers such questions as, *display the last point in the path*, or *what is the next point in the path*. And the blue button will default to the system's. It will control the size, location and existence of the window itself. Figure 6 is Smalltalk-80 code demonstrating this.

```
.....

redButtonActivity
" add the current cursor point to the linked list"
        aView isCollapsed ifTrue:[
                ^ Window is collapsed'
                        displayAt: view viewport origin + (3@3)].
        (aView contains:SensorCursorPoint) whileFalse:[
                ^ invalid point '
                        displayAt: view viewport origin + (3@3)].
        List addLast:(nextLink: SensorCursorPoint).
        CurrentPoint <- SensorCursorPoint.
        List setPoint:CurrentPoint.
        aWindow update.

"-------------------------------------------------"

yellowButtonActivity
|index|
        aView isCollapsed ifTrue:[
                ^ Window is collapsed'
                        displayAt: view viewport origin + (3@3)].
        yellowButtonMenu ~~ nil
                ifTrue:[
                        index <- yellowButtonMenu startUpYellowButton.
                        index ~= 0
                                ifTrue:[
                                  result <- (yellowButtonMessages at:index).
                                  self menuMessageReceiver perform:result.]
                                ifFalse: [super controlActivity]].

"-------------------------------------------------"

run
        self getWindow.
        ScheduledControllers scheduleActive:self.
```

```
"-------------------------------------------------"

initialize
        YellowButtonMenu <- PopUpMenu labels:
        'remove point
        previous point
        next point
        first point
        last point
        exit' lines: #(1 2 3 4 5).
        YellowButtonMessages <- #(rem prv nxt frst lst ext).

"-------------------------------------------------"

getWindow
        aWindow <- Window new.
        aView <- WindowView new.
        super initialize.
        self
                yellowButtonMenu:YellowButtonMenu
                        yellowButtonMessages:YellowButtonMessages.

"-------------------------------------------------"

rem
"remove current link and point to the next one"
|next |
        next <- List next.
        List remove:CurrentPoint.
        aWindow update.
        CurrentPoint <- next.

"-------------------------------------------------"

prv
"set current point to the point before the current point"
        CurrentPoint <- List previous:CurrentPoint.
        List setPoint:CurrentPoint
        aWindow update.

"-------------------------------------------------"

nxt
"set current point to the point after the current point"
```

```
        CurrentPoint <- List next.
        List setPoint:CurrentPoint
        aWindow update.

"-------------------------------------------------"

frst
" set the current point to the first element of the list "
        CurrentPoint <- List first.
        List setPoint:CurrentPoint.
        aWindow update.

"-------------------------------------------------"

lst
" set the current point to the last element of the list "
        CurrentPoint <- List last.
        List setPoint:CurrentPoint.
        aWindow update.

"-------------------------------------------------"

ext
"exit from this section "
^super exit

"-------------------------------------------------"

.....
```

Figure 6. Defining Points In A Path

Points are stored as a linked list. Each link has two parts, data and a pointer. The data part contains the point in the window and the pointer points to the next link/point in the list/path. When the user *clicks* the red mouse button, the **redButtonActivity** is executed. It will create a link, put the current point (pointed to by the cursor) in the data part and point the last link of the list to the new link.

The **yellowButtonActivity** is executed when the user *clicks* the yellow button. A

menu is displayed as shown in Figure 7.

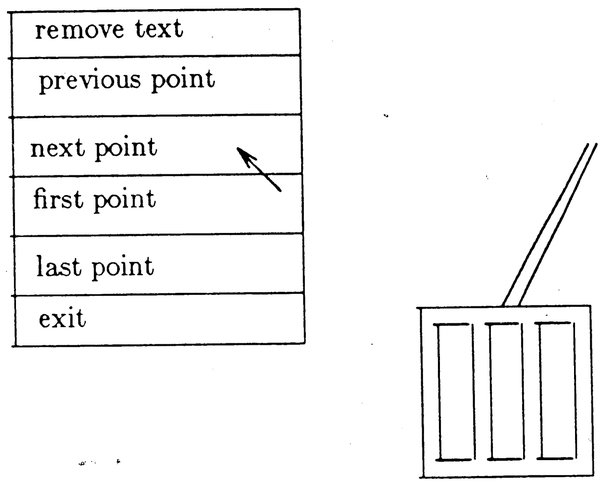


Figure 7. Yellow Button Menu

Depending on the menu item selected:

— The current point (marked by the **X**) is removed from the path.

— The point in the path that precedes the current point becomes the current point.

— The point in the path that follows the current point becomes the current point.

— The first point in the path becomes the current point.

— The last point in the path becomes the current point.

— The user exits from this section of the program.

4.2 Structured Programming

Smalltalk is considered a structured programming language/environment. The notion of *methods* is equivalent to subroutine and procedure calls. Data structures are defined as such objects as arrays, linked lists, dictionaries, sets, bags, collections, and so on. They are accessed and manipulated in similar fashion as those of other structured languages. Parallel processing, semaphores and interaction with external devices are easily implemented.

Perhaps one of the most unique features of Smalltalk is its hierarchy of classes. Each class is a sub class of another, thus inheriting properties of its super class. This layout can be defined as *automatic* or *pre-defined* structure. If a class defines a *method* having the same name as one defined by its super class, the system does not get upset, as with other structured languages. On the same token, the system will not bark if the class uses a *method* it had not defined. The *method* is assumed to be defined in the super class, thus program control is shifted upwards until the definition is found.

4.3 State Table Programming

State tables in Smalltalk can be defined as a dictionary. The key to the dictionary is the item which causes a state change. Consider the finite state automaton of Figure 8.

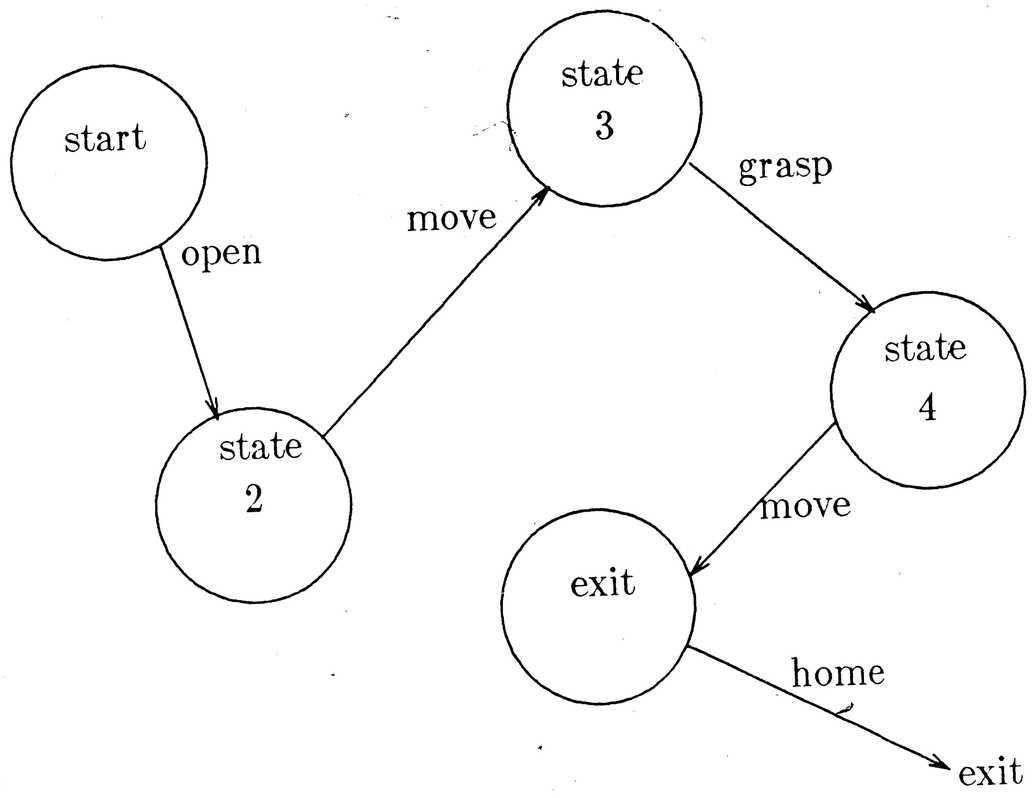


Figure 8. Finite State Automaton

Assume this represents a series of steps to move a part to a specified location. For simplicity, assume there is one possible part to be moved and only one possible place it can be moved to. According to the Figure, if the current state is 2 and a request comes through from the user to move the robot to the part, the state changes to 3 and the move method is executed. Also, the Figure shows no other inputs from the user are allowed and there is only one state that can follow after state 2. Figure 9 demonstrates a segment of code to support this.

```
.....

newState <- Dictionary new.
newState at:#start put:#2.
newState at:#2 put:#3.
newState at:#3 put:#4.
newState at:#4 put:#exit.
newState at:#exit put:#error.

execute <- Dictionary new.
execute at:#start put:open.
execute at:#2 put:moveToPart.
execute at:#3 put:graspPart.
execute at:#4 put:moveToPlace.
execute at:#exit put:home.

input <- Dictionary new.
input at:#start put:#open.
input at:#2 put:move.
input at:#3 put:grasp.
input at:#4 put:move.
input at:#exit put:home.

State <- #start.
Input <- self getFromUser.

[ true ] whileTrue:[
        ((State printString) = 'exit') ifTrue:[ ^self ].
        (input at:State = Input)
                ifTrue:[ perform execute at:State.
                        State <- newState at:State.
                        Input <- self getFromUser.
                       ]
                ifFalse:[self error:(input at:State)].

]

.....
```

Figure 9. State Table Implementation

Three separate dictionaries store the next state, valid input to the state and the method to execute, if the input matches that of the user. Each one is indexed via

the current state (class variable *State*). The method **getFromUser** prompts the user for a command to be sent to the robot. The command is edited and checked for validity. It is then stored in the class variable *Input*. This variable is checked against the valid input for the current state. If not valid, the **error** method is executed. On the other hand, if it is a valid input, the proper method for that state is executed and the state is changed to the state in the *newState* dictionary.

The methods for **getFromUser** and **error** are not shown in this example. They interface with the user, prompting for information.

4.4 Task Oriented Programming

Task Oriented Programming allows the user to direct a robot as they would direct a person to perform a specified task. For example, the user would like to tell the robot to *pick up the book and place it on the box*. The command executed is short, meaningful and understandable. But the program to carry out the task is very lengthy in size and can be complicated to follow and understand. For example, the Smalltalk-80 command *pick:book place:box* is implemented as shown in Appendix I. The program uses two robot arms to pick up an object and place it at a specified destination. The two arms are coordinated in such a way that if the arm picking up the object is not able to reach the specified destination, it places the part in the center of the plane. The second arm then picks the part up from the center and places it at the final destination.

The program is intense and takes most all situations into account. In the process of transferring the part to its destination, a path is chosen such that the arm travels

parallel to the x-y plane. It travels at a height of one plus the height of the tallest object that intersects the x-y plane of the object in question and the arm currently moving. This guarantees a sure path, if the arm has the capabilities of reaching that height.

The plane is referred to as the x, y and z area that holds the items. Valid items of the plane are two robot arms, a box, a chair and a book. See Figure 10 for their locations with respect to each other.

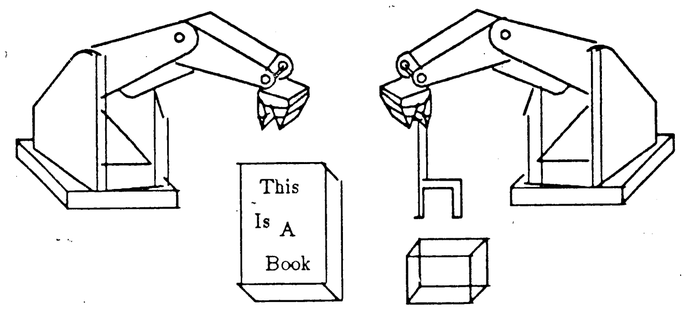


Figure 10. The Objects Of The Plane And Their Locations

The program consists of one class, *Task*, which is a sub class of the system's class *Object*. Class variables are those variables which define the part being transferred (the current part), the desired destination (the current destination/place) and the two robot arms involved in the transferring of the object. The part and place variables are taken from the command line and the robot arms are fixed items in the plane. This project uses two robot arms named arm1 and arm2. There are three objects defined within the plane that can be transferred, box, chair and book.

The database will consist of information defining the following:

- x, y and z coordinates for the gripper of the two arms and the three objects.

- Which arm can reach which object.
- For each arm and object, a gripper width is defined. Meaning, the gripper should be able to open as wide as the defined values in order to grip the objects.
- For each item in the plane, a flag is set to designate if the object is mounted to the plane or not.
- A maximum reach height is defined for each arm.

The ArmCarry table is initialized to nil. This table holds the name of the object the arms are currently carrying. selected from a database. The class variables are defined as follows:

— *ArmPick*. The robot arm that will move the part from its current location and place it either at the center of the plane or at the final destination.

— *ArmPlace*. The robot arm that will move the part to the final destination. If *ArmPick* is the same as *ArmPlace* one arm is used in the task.

— *CurPart*. The current part that is being transferred.

— *CurPlace*. The final destination for the part being transferred.

The instance variables will define the dictionaries used as well as the departure height for the moving arm. The program manipulates eight dictionaries as described/defined below. The values stated are the initial/home values for the objects in the plane.

— *Reach* holds the name of the arm which can reach the object specified by the key. For example, *arm1* can reach *box*.

keys	values
box	arm1

chair	arm1
book	arm2

— *Xaixs, Yaxis Zaxis* dictionaries hold the x, y and z coordinates for the items in the plane. They are random numbers chosen for the sake of this example and do not represent any particular form of measurement.

keys	Xaxis values	Yaxis values	Zaxis values
box	1	3	2
chair	7	5	4
book	8	1	1
arm1	5	6	5
arm2	3	1	7
center	6	3	0

— *Grip* dictionary implies the object can be gripped and the arm gripper must to be at least *value* wide to successfully grip the object.

keys	values
box	4
chair	2
book	2
arm1	3
arm2	1

— *Mount* determines if the object is mounted to the plane or not. A value of 1 means it is mounted while the a 0 value means it is not.

keys	values
box	1
arm2	0
arm1	0
chair	1
book	1

— *MaxHeight* is the maximum height the arm can reach.

keys	values

arm2	20
arm1	6

— *ArmCarry* holds the name of the objects the arm is currently carrying. A value of *nil* means the arm is not carrying anything.

keys	values
arm1	nil
arm2	nil

The program makes use of 12 methods broken up into 2 different categories.

arm controller

This category of methods is responsible for making arm decisions such as: which arm will be used to move the object? will one or two arms be used in the movement? which path will be taken? It supports methods involved in the actual movement of the arm and its gripper. The following methods are defined in this category.

— *ArmSelect* selects an arm to pick up the object and one to place the object at the specified location. It searches the *Reach* database with a key specified by *CurPart*. If an entry is found, it assigns the class variable *ArmPick* the name of the closer arm. It then repeats the same process for *CurPlace* and *ArmPlace*. After selecting the arms, it determines if the part is mounted to the plane or not. If not, it saves the initial position of the two arms, the part being moved and the final destination in instance variables. That is, the x, y and z coordinates for *ArmPick*, *ArmPlace*, *CurPart* and *CurPlace* are saved in instance variables.

— *selectPath:arm destination:part* determines the highest point on the z axis and stores it in the instance variable *DepartHeight*. If this point is greater than the maximum height the arm can reach, it calculates the greatest height for all objects that intersect the x-y plane of the arm and that part.

— *approach:arm where:object* will change the x, y and z coordinates of the arm argument to those of the part (passed as the object argument). The z coordinates are set to one more than the z coordinate of the part. If the arm is currently carrying a part, the coordinates of the part will also be updated.

— *grip:arm* the robot arm represented in the *arm* argument passed to this method will grip the part represented by *CurPart*. Gripping is accomplished by moving the arm down to the z axis of the part (it was previously positioned above the part), closing the gripper and placing the name of the part in the *ArmCarry* database using the key specified by *arm*.

— *tradeOffPut* This method will move the *CurPart* to the center position of the plane or to the specified place, depending on the values of *ArmPick* and *ArmPlace*. That is, if *ArmPick* is not equal to *ArmPlace*, the part is moved to the center of the plane. Otherwise, it is moved to the original destination.

Action is not carried out in this method per say. It chooses the proper arguments then executes the *approach* method.

— *tradeOffGet* is similar to *tradeOffPut* in that it works from the center of the plane. It is executed only when *ArmPlace* is not equal to *ArmPick*. It selects a path for ArmPlace to travel from its current position to the center of the plane. *Approach* is executed to move the arm to the center and the part is gripped. A path is then selected for the arm to travel from the center to the specified destination. *Approach* is executed to move the arm to the location and *release* is executed to release the part from the arm. *Home* is called to move both robot arms to their initial states.

— *release:arm object:part newZ:z* takes an object and moves it to its calculated Z axis (*newZ*) It then updates the database indicating the arm is no longer carrying a part.

— *depart:arm* changes the z coordinate of the arm to that of *DepartHeight*. It searches the database (using *arm* as a key) to see if the arm is carrying a part. If so, it changes the z coordinate of the part as well.

start it all up

This category of methods will initialize the database, describe the task to the user, control the transferring of the part, put the arms in their initial (home) states and exit. The following is a description of the methods.

— *home:arm xx:x yy:y zz:z* is called with the x, y and z coordinates for the arm. These coordinates are the coordinates for home position of the arm. The method will update the database by assigning the arm coordinates these values.

— *initialDescription* prints a message to the user stating which part will be transferred to what location and which arms are involved in the transferring.

— *initialize* defines and initializes the databases/dictionaries used.

— *pick:part place:place* oversees the task. It is executed when the user makes a *pick and place* request. *Part* and *place* are literals passed into this method from the user. They are converted to lowercase letters and symbols so they can easily be used as keys for the dictionaries. The *initialize* method is called to set up the databases. *ArmSelect* is called to see if the *part* and *place* are valid items in the plane and to select the arms used in this task. *SelectPath* selects a path from the current position of the arm to the part. Next *Grip* is executed to instruct *ArmPick* to grip *CurPart*. The arm is instructed to release the part and is finally moved to its home position.

4.5 Advantages and Disadvantages Of Incorporating The Four Levels

Advantages

— A robot program is a path in space composed of multiple points. At each point in the program, a user indicates whether the robot is to stop or pass through the point. Smalltalk-80 incorporates that capability through the use of menus. Decision making can be as simple as *clicking* the *yellow* button.

— The graphic animation, which displays hidden lines, can be used to select an appropriate arm or to lay out the plane. The animation also facilitates the planning of robot programs and can be used to a limited extent for visual collision detection.

— Debugging is faster and more accurate. Most robot arms lose accuracy when moving from one point to another. This is eliminated since precise

points are used in the place of relative points.

Disadvantages

— Smalltalk is not clearly understood. Since it is an object oriented language, it spends its time sending messages to objects and unless you realize which is the message and which is the object, you are lost in the logic or flow of the language. So, yes it is structured; but that does not imply it is easy to follow the logic of the program.

— Sensing capabilities are limited to the accuracy of the cursor. Sensing can be very well displayed in a simulated system as opposed to actual systems. Sensing is a real-time function that cannot be pre-defined.

— To implement state table programming, the dictionary must have an entry for every possible input to a state. This can become large and unmanageable.

5. Chapter 5: Simulated Application

5.1 Task Definition

The sample program in Appendix I is linear in design. Each step logically follows the next. Even though it is implemented in Smalltalk, it does not take full advantage of the object oriented approach to programming. This chapter takes another look at *pick and place* but this time defining each object of the plane as an *object.*

The task involves one or two robot arms moving objects from one place to another. The task is carried out by one of the robot arms in the plane. The plane consists of two arms, a book and a box. Modularity is preserved by defining each object in the plane as a Smalltalk-80 object representing a class. The addition and subtraction of an object to the plane requires a class be added to or subtracted from the system. This example, though not thorough and complete in design, produces a shell for future expansions. It serves as a sample approach when using Smalltalk-80 to simulate robot programs.

Path Planning:

Due to limiting capabilities of the graphics display screen, the plane used in this example is x-y coordinates and not x-y-z coordinates. Therefore, path planning consists of finding a point high enough on the y axis that the arm can reach without colliding with other objects. If such a point is found, the arm moves parallel to the x-axis until it reaches its destination.

Collision:

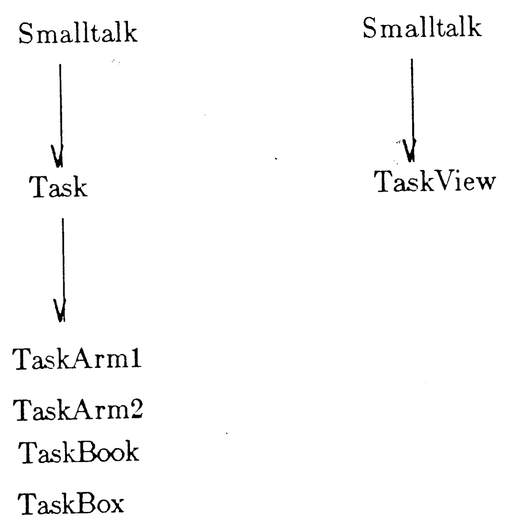
The program used in this example is not advanced in collision avoidance. It recognizes a possible collision before the objects actually make their moves. If one is detected, task execution simply halts, returning all objects back to their home positions.

Adding and Removing Objects:

Each object of the plane is a subclass of class `Task`. As such, they respond to a common/standard interface. If an object is to be added to the system, the minimum requirement is that it conforms to the interface.

5.2 Class Definition

There are six classes defined in this example, as shown below.



Task View:

Class `TaskView` is a subclass of the Smalltalk-80 class `StandardSystemView`. It is responsible for managing the view that the objects are displayed on. Each instance of `StandardSystemView` has an instance variable *model*. This variable defines the connection between the display view and the objects being displayed. The connection is one way, as shown in Figure 11, and is established by sending the message *model:* to the display view.

The view is displayed when the instance receives a message *displayView, update, update:* or *display*. In either case, *displayView* is the method executed. There, `TaskPlane` is enumerated, sending a message of *display* to all objects defined in it.

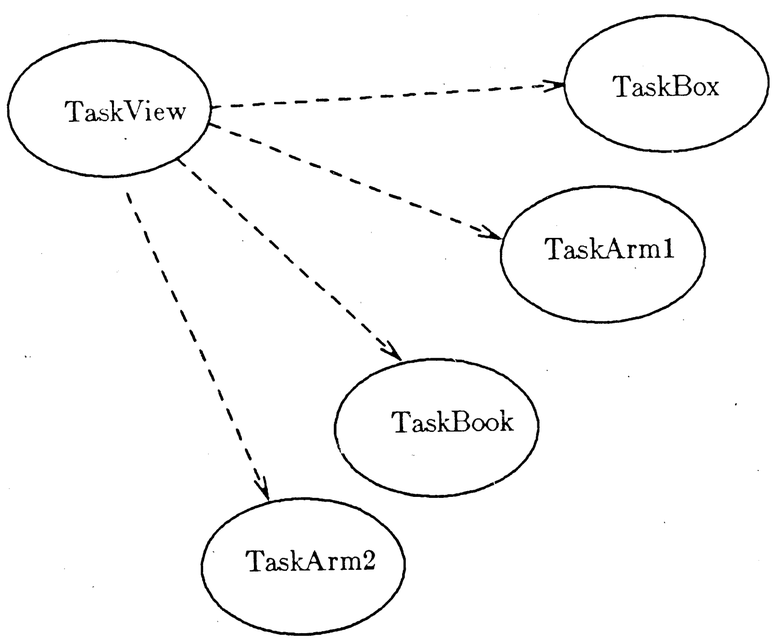


Figure 11. Connection From TaskView To Other Objects

Task:

Class `Task` is a subclass of the Smalltalk-80 class `Object`. It is the controlling class of the executed task. Instances are created via class methods `new` or `pick:place:`. The `new` method for the super class is executed and then some initialization is performed as follows:

— An instance is created for each object of the plane. Their instances are stored in the class variable `TaskPlane`. TaskPlane is a dictionary, where keys are the names of the objects in the Defining the pool dictionary `TaskPlane` as a dictionary is in support of modularity. If an object is added to the system, dynamic allocation of another entry in the dictionary for this new object is possible. Likewise, the deletion of an object requires an entry be removed from the dictionary.

— An instance of class `TaskView` is created and initialized. This instance is stored in the class variable `AView`. This establishes a one way link from `Task` to `TaskView`, thus updating Figure 11 as shown in Figure 12. The model for the view is set by sending the *model:TaskPlane* message to `AView`. And, the view is displayed by sending the message *displayView* to `AView`.

— Task execution is controlled by a finite state automaton. The state table is initialized by executing the method *initTable*.

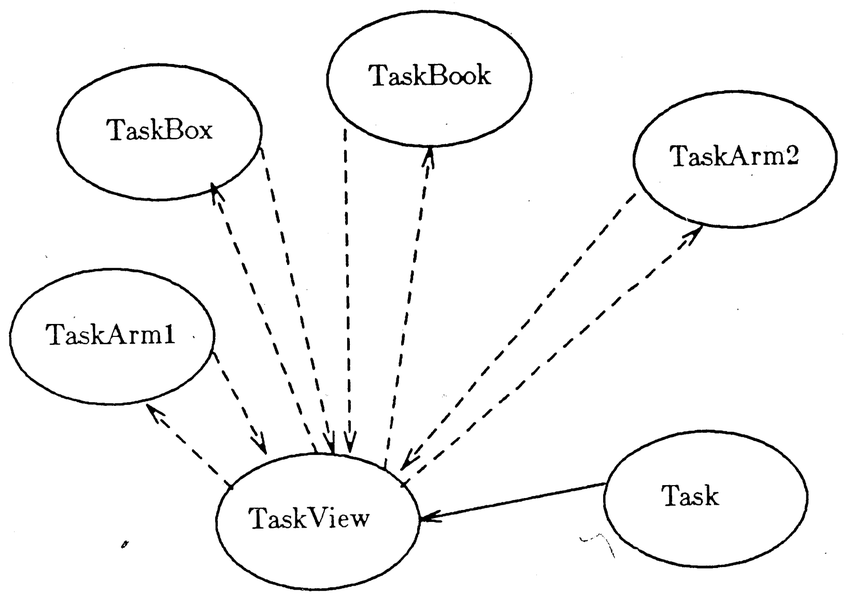


Figure 12. Connection From Task To TaskView

TaskArm1, TaskArm2, TaskBook, TaskBox:

Each object of the plane is a subclass of `Task`. They define, among other things, a graphical view of themselves. Instance variables keep track of location, coordinates and position in the plane relative to the robot arms. Therefore, the objects know how to display themselves and move themselves to another location. They calculate boundary points with respect to the view they are defined in. Also, given points of other objects, they detect intersections, or the lack of.

The objects are connected to the display view by way of a dependency. The dependency is created by adding an entry into a Smalltalk-80 designated dictionary. The addition is performed by sending the message *addDependent:AView* to the super class. The result of this message is shown in Figure 13. There, each object is connected to the display view by way of the dependency dictionary; the

display view is connected to class `Task` by way of class variable `AView`; and the display view is connected to each object by way of the model.

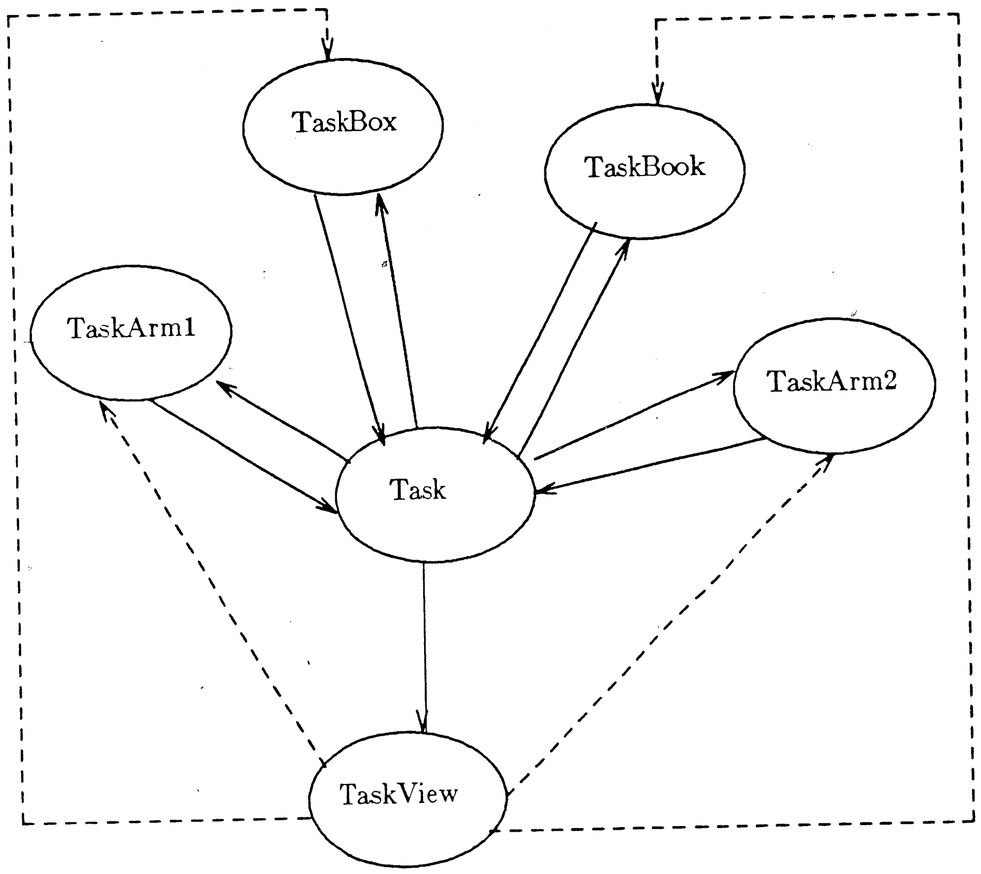


Figure 13. Connection Between All Classes

5.3 State Table

Execution of the *pick:place:* task is controlled by a finite state automaton, as shown in Figure 11 and Appendix 2. Each state represents the execution of a method.

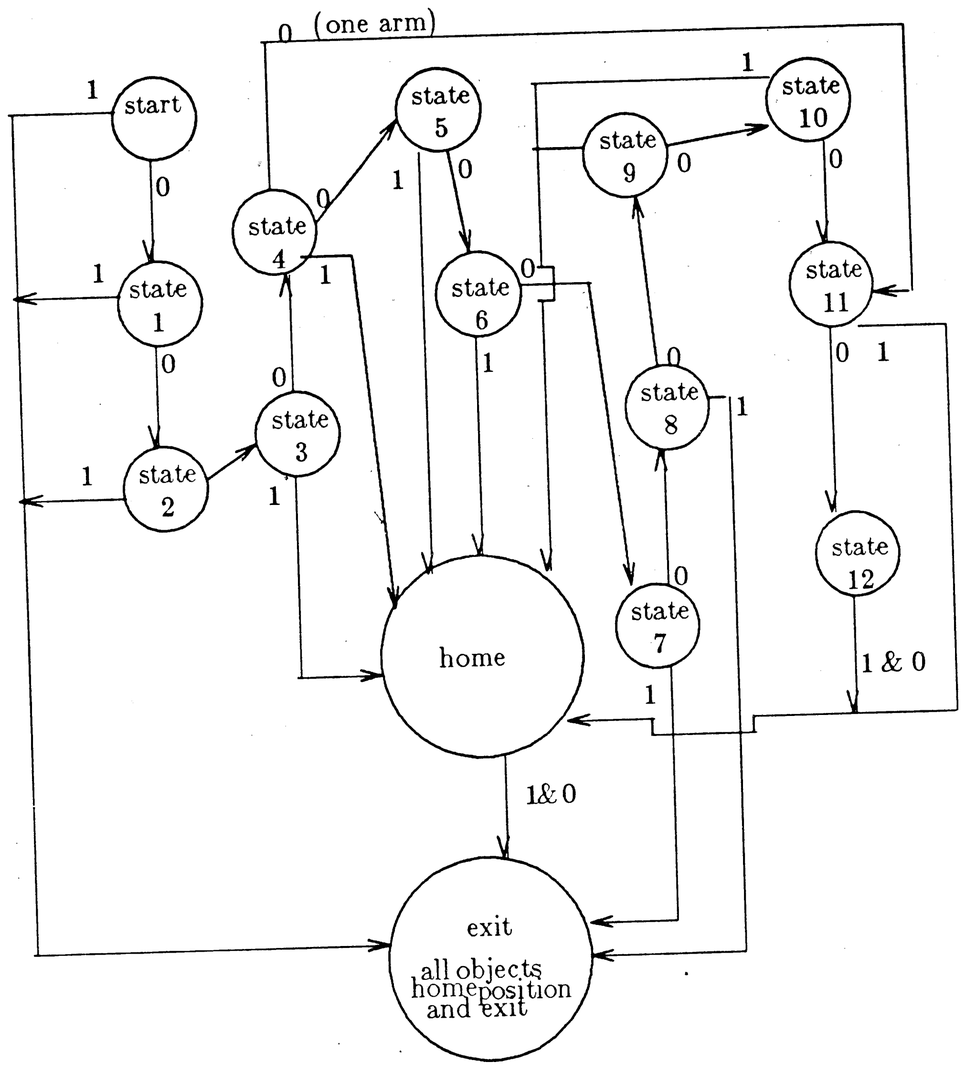


Figure 14. State Table Diagram

Class variable ERROR denotes the execution status of the previous state (0 for success and 1 for failure) and STATE is the next state to be executed. State changes are caused by executing a method defined in the *stateTable* dictionary. The key for the dictionary is the current state suffixed with the value for ERROR. For example, consider the following two lines in the dictionary. If the current state is *state1* and ERROR is set to 0, method *state1* is executed. Otherwise, if ERROR is set to 1, method *exit* gets executed.

```
.....

stateTable at:#state10 put:#state1
stateTable at:#state11 put:#exit

.....
```

Just before exiting from each method, a state change is made by setting class variable STATE to the name of the next state. As long as class variable EXIT is false, *pick:place* loops, concatenating STATE and ERROR to get a key for the dictionary. Then, executing the method associated with that key. EXIT is set to true in method *exit*, after all objects are moved to their home positions.

5.4 Standard Interface

The program is designed such that it is portable and easily expandable. Thus, each object in the plane is independent of any other. Furthermore, the main program is not dependent on the properties of each object. One object can be circular in form, one rectangular and another elliptical. What is required is that object definitions conform to a standard interface program. The interface consists of the

following methods.

display: Responsible for calculating the display coordinates of its graphical form and displaying it such that it is enclosed by the display view. This method is called by the instance of `TaskView` whenever the display view needs to be updated.

new: This method creates and initializes an instance of the object. A typical method definition consists of

```
.....

|tempObject |
tempObject <- super new.
tempObject setValues.
^tempObject

.....
```

However, the definitions are not limited to this. Since each object has different properties, it is not feasible to standardize the content of this method, just as long as each object correctly responds to it.

getMaxX, getMaxY: These methods are basically used in path planning when deciding the maximum height the arm should reach to avoid collision. They represent the x and y coordinate of the heighest point of the object.

getName: This method returns the name of the object as a character string.

getGrip: Returns the x-y coordinates at which the object should be gripped by the arm. Typically it is the center point of the object or it is the upper surface.

mounted: This boolean returns true if the object is mounted to the plane and cannot be moved. It will otherwise return false.

isArm: This boolean will return true for all robot arms and false otherwise.

approach:point: This method makes sense if the receiver is a robot arm or if the it is being carried by an arm. It determines the maximum traveling height which the arm will follow. The arm is positioned at the location denoted by *point*. Since it involves motion, it must add the `change` message in the definition of the method. This message is sent to the display view, denoting an update of the display is required.

home: The method responsible for placing the object at its initial coordinates.

canReach:point: This message, when sent to a robot arm, checks to see if the arm can be extended to the location specified by *point*. It creates a reference point by adding its center location to its reach points. If the location in question, *point*, falls within the reference point, this boolean returns true. Otherwise, the method returns false. Since arms are the only objects in the plane that can reach out to some other object, this message returns false when sent to objects that do not represent arms.

release: This method has two meanings depending on the receiver of the message. If the receiver is a robot arm, the gripper of the arm is opened and the status of the arm is updated to denote it is no longer gripping an object. The message is then sent to the object carried by the arm, to also update the status of the object.

depart: Like *release*, this method has two meanings. When sent to an arm, it moves the arm upward along the y axis. It then sends a message to the object the arm is carrying, if any. This message updates the location and position of the object with respect to the moving arm.

6. Chapter 6: Summary

6.1 Summary

The major objective of this document was to analyze the Smalltalk-80 environment for use in robotic programming. To achieve this objective, Smalltalk-80 and existing languages were compared to an ideal language. The characteristics and qualities of this ideal language were introduced. Numerous languages were analyzed to see how they would fit into this model. They were categorized into four levels, depending on language emphasis. The advantages and disadvantages of these levels on robot programming were stated.

Segments of programs demonstrated how the Smalltalk-80 environment incorporated each of the four levels. Smalltalk-80 was then analyzed more closely emphasizing a simulated example as opposed to the actual programming of the robot. In addition to supporting the four levels, Smalltalk-80 brought something new to the idea of robotics, namely an object oriented way to program them.

6.2 Research Issues

Now that it has been established that Smalltalk-80 adequately simulates robot programs, additional research projects need to be undertaken. Future research related to the use of Smalltalk-80 for robot programming are given below. The design and implementation of these features is beyond the scope of this paper.

Collision Avoidance

Collision avoidance is still an open issue in robot programming. The introduction of another program language or environment surfaces this issue again. Graphics, on the other hand, can be used for visual collision detection. This is not recommended because collision could occur and not be observed by the user. Research is ongoing to develop heuristic methods of detecting collisions. For example, researchers at Lehigh University[9] have surrounded the hand of the robot with a sphere and checked for interference between the sphere surrounding the hand and the objects in the modeled system. When interference is detected, the actual computations are made to determine whether the hand or any object it may be holding is, in fact, colliding with something else in the workspace.

This is a good heuristic but not complete. One can easily have collisions between the workspace and parts of the robot that do not involve the hand or its surrounding sphere. For example, robots with elbows could bang their elbows into parts of the workspace and this would not be detected. This process of putting a sphere around the elbow and other places likely to be involved in collisions can be continued, and work is progressing along these lines. However, researchers are now looking into methods of detecting all interferences to more accurately detect collisions in the robot's world.

Even though this heuristic is simple enough to implement in Smalltalk-80, what about re-planning the robot's path in the event collision is detected? This requires investigation into the use of a world model. At the time of

collision, a dynamic view of the workspace needs to be available to the system. This implies the use of a dynamic database and the capabilities to dynamically change the robots positions. How can this database be used to avoid collisions?

Standard Interface

The Smalltalk-80 environment is defined on top of an operating system resembling UNIX®. As such, it has access to a shell interpreter language and the I/O ports of the machine. Language commands for a particular robot can be interspersed with the graphic instructions and down-loaded to the robot through the output ports. The key benefit of such a system is that the user can see a graphic animation of the working robot workspace.

Because of the lack of standard robot interfaces, a system cannot produce a robot program unless special arrangements have been made between the system designer and robot expert. Will there exist a standard robot interface? If so, does Smalltalk-80 have the capabilities to support such a standard?

Dynamic Path

In the "real world", many tasks are performed at once. Some dependent on another and some not. Timing and efficiency becomes extremely important in the system's design. A breakdown or malfunction of the system resembles falling dominos: one thing leads to another, and, before you know it, there is a chain reaction.

If collision was detected in the course of arm motion, halting the arm may not be wise. Alternate paths may be a possibility in the event one path becomes blocked. There are two ways to alter a path. The paths can be pre-defined in an array or created dynamically. Dynamic construction is the more natural and efficient way. Pre-declaration places dependencies on the current make up of the "world". This area needs to be further investigated and implemented, if possible.

Sensors

As shown in the example of chapter 5, each item in the plane is defined as a form enclosed by a rectangle. This design limits the sensing capabilities to the surrounding box. For some applications, this may be adequate. Others may require more precise and accurate measurements. Methods simulating sensor behavior, particularly for complex vision and taction sensors need to be developed.

Machine-Machine Interface

A challenge in the development of advanced robotic software is the machine-to-machine interface, a problem which has been largely ignored to date. This is a problem with current versions of second generation languages because flexible manufacturing systems and islands of automation cannot be integrated into higher-level entities without the required hardware and software interfaces. Convenient means must be found to interface such systems as Smalltalk-80 with hierarchical controllers and complex sensors.

Easy ways must be created for interfacing Smalltalk-80 with the machine tools, inspection stations, and material handling systems.

Natural Language Understanding

It would be ideal to program a robot with a task much the same as one would describe the task to a person. Researchers are trying to make it possible for robot control systems to read natural language input describing a task and then act upon them accordingly. For example[9]:

> **Mate the part with the hole in it with the part with the peg in it, so that the peg and hole are aligned, and the corners of the surface are aligned.**

It is clear this requires parsing and interpreting complex sentences. Not only is work needed in the area of natural language understanding, but in the area of task-decomposition. The 3 lines of text above need to be decomposed such that the manipulators of the robot can understand and act accordingly. Parse and state tables are easily implemented in Smalltalk. Is it feasible to make use of their implementation and design a natural language understanding system around them?

6.3 Conclusion

Eventually, robot programming may become so intelligent that it routinely models its environment and its behavior in order to anticipate and avoid collisions, deal with uncertainty, and generally reason about its task. When this happens, the simulated system may cease its independent existence and become an integral and

necessary part of robot programming. Smalltalk-80 is the "first generation" of such simulated languages.

REFERENCES

1. **Working Robots** Fred D'lgnazio, Lodestar Books, E. P. Dutton New York
2. **A Comparative Study of RobotLanguages** SusanBonner, Kang G. Shin. COMPUTER vol.15 num. 12, Dec. 1982
3. **MICROBOT InstructionCode Compiler User's Guide and Programming Manual**. Kieth J.Werkman, 1986, Lehigh University Institute for Robotics
4. **AML/X User's Manual (Draft)** Lee R. Nackman, Mark A. Lavin, Russell H. Taylor andWalter C. Dietrich, 1986 IBM TJ Watson Research Center
5. **AI Programming Languages** TheInformation Technology Series, Volume VI, Earl D. Sacerdoti, Richard E. Fikes, Rene Reboh, Daniel Sagalowicz, Richard J. Waldinger and **M. Wilber, 1984**
6. **PROLOG - PROgrammingLOGic** Kieth Werkman, Lehigh University, April,1986
7. **QLISP A Language forthe interactivedevelopment of complex systems**.gThe Information TechnologySeries, Volume VI, Earl D. Sacerdoti, Richard E. Fikes, Rene Reboh, Daniel Sagalowicz, Richard J.Waldinger and B. M. Wilber, 1984
8. **Hierarchical ControlFor Robots In An Automated Factory** J. S. Albus, C. R. McLean, A. J. Barbera, M.L. Fitzgerald, NationalBureau of Standards, 1983
9. **An Analysis of RobotSoftware And Plans For Its Enhancement** Dr. Roger N. Nagel, Scott R. Garrigan, Lehigh University, technicalreport #85-001,June, 1985.
10. **Robot Manipulator Control Using the "C" Language under UNIX** VincentHayward, Richard P. Paul, November 1983
11. **Smalltalk-80, The Interactive Programming Environment** Adele Goldberg, 1984.
12. **PI: A Case Study In Object Oriented Programming** T. A. Cargill, SIGPLAN Notices 21(11), Nov., 1986,p350-60
13. **Automated Programmering-The Programmer's Assistant**, Proceedings of the Fall Joint Computer Conference, Dec. 1972. W. Teitelman.
14. **The Interlisp Programming Environment**, IEEE Computer Magazine, April 1981, pp. 25-33.gW.Teitelman, L. Masinter.
15. **A Simple Motion-Planning Algorithm for General Robot Manipulators**, Tomas Lozano-Perez. IEEE Journal of Robotics and Automation, Vol. RA-3, No. 3, June 1987.

16. **Kinematics of Two-Arm Robots** Ahmad Hemami. IEEE Journal of Robotics and Automation, Vol. RA-2, No. 4, December 1986.

17. **Robot Path Planning using Intersecting Convex Shapes: Analysis and Simulation** J. Sanjiv Singh, Meghanad D. Wagh. IEEE Journal of Robotics and Automation, Vol. RA-3, No. 2, April 1987.

18. **A Type System for Smalltalk: An Optimizer's View** Ralph Johnson, University Of Illinois, May 1987.

19. **Smalltalk-80, The Language And Its Implementation** Adele Goldberg, 1984.

20. **Model-Based Programming and Control of Robot Manipulators** Charles N. Stevens, General Motors Research Laboratories, COMPUTER, August 1987.

21. **The Impact of Robotics On Computer Science** John E. Hopcroft, COMMUNICATIONS of the ACM, June 1986.

Appendix I: Linear Programming In Smalltalk-80

```
Object subclass: #Task
        instanceVariableNames: 'MaxHeight Reach Xaxis Yaxis Grip Mount Zaxis
                DepartHeight ArmCarry '
        classVariableNames: 'ArmPick ArmPickX ArmPickY ArmPickZ ArmPlace
                ArmPlaceX ArmPlaceY ArmPlaceZ CurPart CurPartX
                CurPartY CurPartZ CurPlace CurPlaceX CurPlaceY
                CurPlaceZ '
        poolDictionaries: ''
        category: 'Appendix-I'!


!Task methodsFor: 'arm controller'!

approach:arm where:object
|carry|
" Arm is the robot arm doing the approaching and object is
  the location it will approach."

self depart:arm.
Xaxis at:arm put:(Xaxis at:object).
Yaxis at:arm put:(Yaxis at:object).
Zaxis at:arm put:DepartHeight.

" see if the arm is carrying something, if so, also change its coordinates."
((ArmCarry at:arm) ~= nil) ifTrue:[carry<-ArmCarry at:arm.
Xaxis at:carry put:(Xaxis at:object).
Yaxis at:carry put:(Yaxis at:object).
Zaxis at:carry put:DepartHeight]!


armSelect
|returnCode flag1 flag2|
" Look up the Reach table for the key specified by CurPart and CurPlace. If
  found, look up the mount table to see if the values at those locations are 1,
  if so the part can be moved. If not, print error and return -1 to designate
  error.
"
flag1<-0. flag2<-0. returnCode<-0.
ArmPick <- Reach at:CurPart ifAbsent:[flag1<- -1].
ArmPlace <- Reach at:CurPlace ifAbsent:[flag2<- -1].
(flag1 ~= -1) ifTrue: [flag1<- Mount at:CurPart ifAbsent:[flag1<- -1]].

(flag1 < 1) ifTrue: [returnCode<- -1].
```

```
(flag2 < 0) ifTrue: [returnCode<- -1].

"Save original state of ArmPick, ArmPlace, CurPart and CurPlace"
ArmPickX<-Xaxis at:ArmPick.
ArmPickY<-Yaxis at:ArmPick.
ArmPickZ<-Zaxis at:ArmPick.
ArmPlaceX<-Xaxis at:ArmPlace.
ArmPlaceY<-Yaxis at:ArmPlace.
ArmPlaceZ<-Zaxis at:ArmPlace.
CurPlaceX<-Xaxis at:CurPlace.
CurPlaceY<-Yaxis at:CurPlace.
CurPlaceZ<-Zaxis at:CurPlace.
CurPartX<-Xaxis at:CurPart.
CurPartY<-Yaxis at:CurPart.
CurPartZ<-Zaxis at:CurPart.
^returnCode!


depart:arm
|carryingPart|
" change the z coordinates of the arm and the
  object it's carrying to the DepartHeight.
"
carryingPart<-ArmCarry at:arm.
(carryingPart ~= '') ifTrue:[Zaxis at:carryingPart put:DepartHeight].
Zaxis at:arm put:DepartHeight.!


grip:arm
" CurPart is gripped by ArmPick. Put the name for CurPart into
  the ArmCarry dictionary for the ArmPick.
"
Xaxis at:arm put:(Xaxis at:CurPart).
Yaxis at:arm put:(Yaxis at:CurPart).
Zaxis at:arm put:(Zaxis at:CurPart).
ArmCarry at:arm put:CurPart.!


release:arm object:part newZ:z
" Take an object and move it to its calculated z axis then
  mark the arm as no longer carrying it.
"
```

```
Zaxis at:part put:z.
ArmCarry at:arm put:nil.!


selectPath:arm destination:part
|x y collide|

" Determine the heighest point on the z-axis and store it in
  DepartHeight. If height is greater than the maximum height
  the arm can reach,  calculate the greatest height for all
  objects that intersect the x-y plane of the arm. If still
  greater than the max height for arm, return error.
"
DepartHeight<-0.
Zaxis do: [:height |(height  > DepartHeight) ifTrue: [DepartHeight<-height]].
DepartHeight<-DepartHeight+1.
(DepartHeight <= (MaxHeight at:arm)) ifTrue: [^DepartHeight].
"else, check each object in the path of the arm."
DepartHeight<- -1.

Xaxis keysDo:
[:key |x <- Xaxis at:key.    " if there is an object intersecting the x axis..."
(
  (
    (x <= (Xaxis at:arm)) and:
    [x >= (Xaxis at:part)]
  )
or:
  [
    (x >= (Xaxis at:arm)) and:
    [x <= (Xaxis at:part)]
  ]
)
    ifTrue: [((Zaxis at:key) > DepartHeight)  ifTrue: [DepartHeight<-(Zaxis at:key) +1.
                        collide<-key.]]
].
 Yaxis keysDo:
[:key |y <- Yaxis at:key.    " if there is an object intersecting the y axis..."
(
   (
     (y <= (Yaxis at:arm)) and:
     [y >= (Yaxis at:part)]
   )
or:
```

```
    [
      (y >= (Yaxis at:arm)) and:
      [y <= (Yaxis at:part)]
    ]
)
 ifTrue: [((Zaxis at:key) > DepartHeight)  ifTrue: [DepartHeight<-(Zaxis at:key) +1.
                        collide<-key]]
].

(DepartHeight > (MaxHeight at:arm)) ifTrue:[DepartHeight<- -1]
^DepartHeight!
```

```
tradeOffGet
|returnCode|
"
  If ArmPick ~= ArmPlace, move CurPart to the center, via ArmPick.
"
returnCode<-0.
returnCode<-self selectPath:ArmPlace destination:#center.
(returnCode = -1) ifTrue:[^-1].
self approach:ArmPlace where:#center.
self grip:ArmPlace.
returnCode<-self selectPath:ArmPlace destination:CurPlace.
(returnCode = -1) ifTrue:[^-1].
self approach:ArmPlace where:CurPlace.
self release:ArmPlace object:CurPart newZ:(CurPartZ + CurPlaceZ).
self home:ArmPlace xx:ArmPlaceX yy:ArmPlaceY zz:ArmPlaceZ.
^returnCode!


tradeOffPut
(ArmPick ~= ArmPlace)
   ifTrue: [self approach:ArmPick where:#center]
   ifFalse: [self approach:ArmPick where:CurPlace]
!!
```

```
!Task methodsFor: 'start it all up'!

home:arm xx:x yy:y zz:z
" Return arm to its home position."
Xaxis at:arm put:x.
Yaxis at:arm put:y.
```

```
Zaxis at:arm put:z.
!


initialize

" The parts of the plane are defined in the dictionaries Reach, Grip,
  Mount, etc. The keys to the dictionaries is the object's name. "

Reach <- Dictionary new.          " which arm can reach the part"
#(box chair book )
        with: #(arm1 arm1 arm2)
        do: [:key :value |Reach at:key put:value].

Xaxis <- Dictionary new.          " the x-coordinate of the part"
#(box chair book arm1 arm2 center)
        with: #(1 7 8 5 3 6)
        do: [:key :value |Xaxis at:key put:value].

Yaxis <- Dictionary new.          " the y-coordinate of the part."
#(arm2 book box chair arm1 center)
        with: #(1 1 3 5 6 3)
        do: [:key :value |Yaxis at:key put:value].

Zaxis <- Dictionary new.          " the z-coordinate of the part. Also the height "
#(box arm2 center arm1 chair book)
         with: #(2 7 0 5 4 1)
         do: [:key :value |Zaxis at:key put:value].

Grip <- Dictionary new.    "this says the object can be gripped and the gripper needs"
#(box arm2 arm1 chair book) " to be at least this wide."
          with: #(4 2 2 3 1)
          do: [:key :value |Grip at:key put:value].


Mount <- Dictionary new.    "0 means the object is not mounted and 1 means it is"
#(box arm2 arm1 chair book)
          with: #(1 0 0 1 1)
          do: [:key :value |Mount at:key put:value].

MaxHeight <- Dictionary new.    " Maximum height the arms can reach"
#(arm2 arm1)
        - with: #(20 6)
          do: [:key :value |MaxHeight at:key put:value].
```

```
ArmCarry <- Dictionary new.    " The objects the arm is currently carrying. "
#(arm2 arm1)
        with: #(nil nil)
         do: [:key :value |ArmCarry at:key put:value].!


pick:part place:place
|returnCode|
"
        Algorithm....
    1  Find an arm to initially move the object.
    2  Find an arm that reaches the final destination.
    3  If no success in finding the arms, return error (-1).
    4  If the part is mounted, print error.
    5  Call selectPath to get a path along the z-axis.
    6  Call approach to approach the arm.
    7  Grip the object.
    8  Select a path to the final destination.
    8  Release the object.
    9  Put arms in home position.
"

CurPart<-(part asLowercase) asSymbol.
CurPlace<-(place asLowercase) asSymbol.
self initialize.
returnCode<-self armSelect.
(returnCode =-1) ifTrue:[^-1] .

returnCode<-self selectPath:ArmPick destination:CurPart.
(returnCode =-1) ifTrue:[^-1] .

self approach:ArmPick where:CurPart.
self grip:ArmPick.
self tradeOffPut.
self release:ArmPick object:CurPart newZ:CurPartZ.
self home:ArmPick xx:ArmPickX yy:ArmPickY zz:ArmPickZ.

(ArmPick ~= ArmPlace) ifTrue: [returnCode<-self tradeOffGet].
^returnCode
```

Appendix II: Simulating Robot Programming In Smalltalk-80

```
Object subclass: #Task
        instanceVariableNames: 'TaskPlane Center Exit Pick Place Arm
          ERROR STATE stateTable '
        classVariableNames: 'AView '
        poolDictionaries: ''
        category: 'Appendix-II'!


!Task methodsFor: 'initialization'!

initPlane
"Create an instance for each item in the plane"

TaskPlane <- Dictionary new.
TaskPlane at:#box put:(TaskBox new).
TaskPlane at:#arm1 put:(TaskArm1 new).
TaskPlane at:#book put:(TaskBook new).
TaskPlane at:#arm2 put:(TaskArm2 new).
TaskPlane at:#center put:(TaskCenter new).

(TaskPlane includesKey:Pick) ifFalse:[Exit<-true].
(TaskPlane includesKey:Place) ifFalse:[Exit<-true].

initTable
"Initialize the state table"

stateTable <- Dictionary new.
stateTable at:#start0 put:#start.
stateTable at:#state10 put:#state1.
stateTable at:#state20 put:#state2.
stateTable at:#state30 put:#state3.
stateTable at:#state40 put:#state4.
stateTable at:#state50 put:#state5.
stateTable at:#state60 put:#state6.
stateTable at:#state70 put:#state7.
stateTable at:#state80 put:#state8.
stateTable at:#state90 put:#state9.
stateTable at:#state100 put:#state10.
stateTable at:#state110 put:#state11.
stateTable at:#state120 put:#state12.
stateTable at:#home0 put:#home.
stateTable at:#exit0 put:#exit.
Exit<-false.
STATE<-#start.

stateTable at:#start1 put:#exit.
stateTable at:#state11 put:#exit.
stateTable at:#state21 put:#exit.
stateTable at:#state31 put:#exit.
stateTable at:#state41 put:#home.
stateTable at:#state51 put:#home.
stateTable at:#state61 put:#home.
stateTable at:#state71 put:#home.
stateTable at:#state81 put:#exit.
stateTable at:#state91 put:#home.
stateTable at:#state101 put:#home.
stateTable at:#state111 put:#home.
stateTable at:#state121 put:#home.
stateTable at:#home1 put:#home.
stateTable at:#exit1 put:#exit.
```

```
ERROR<-0.

initView
"initialize the display view"
|frame |

AView<-TaskView new.
AView label:'Pick And Place Simulation'.
AView insideColor:Form white.
frame<-(400@400) extent:(480@300).
AView window:frame viewport:frame.
AView borderWidth:5.
AView model:TaskPlane.
AView display.

!Task methodsFor: 'states'!

start
"Clear the ERROR flag and set STATE to state1"
ERROR<-0.
STATE<-#state1.


state1
"Identify the arm to reach Pick and specify if
 center will be used for trade off"

Arm<-''.
Center<-false.
ERROR<-0.
STATE<-#state2.

"replace class variables Pick and Place with instances"
Pick <- TaskPlane at:Pick.
Place <- TaskPlane at:Place.

TaskPlane
   associationsDo:[:each|(each value isArm)
                        ifTrue:[(each value canReach:(Pick getGrip))
                     ifTrue:[(Arm == '')
                  ifTrue:[Arm<-each key]
               ]
            ]
         ].
```

```
"see if will need one or two arms"
(Arm ~= '')
  ifTrue:[((TaskPlane at:(Arm asSymbol)) canReach:(Place getGrip))
           ifFalse:[Center<-true]]
  ifFalse:[ERROR<-1].


state2
"Have Arm approach the center of Pick"
|point|

point<-Pick getGrip.
ERROR<-(TaskPlane at:Arm) approach:point.
STATE<-#state3.

state3
"Have Arm grip Pick"

ERROR<-(TaskPlane at:Arm) grip:Pick.
STATE<-#state4.

state4
"Have Arm depart while carrying Pick"

ERROR<-(TaskPlane at:Arm) depart.
Center ifTrue:[STATE<-#state5]
       ifFalse:[STATE<-#state12].

state5
"Put Pick at the center and wait for the next arm
 to pick it up and place it at the final destination"
|point|

point<-(TaskPlane at:#center) getGrip.
ERROR<-(TaskPlane at:Arm) approach:point.
STATE<-#state6.

state6
"Release Pick at the center location"

ERROR<-(TaskPlane at:Arm) release.
STATE<-#state7.

state7
"check to see if a second arm can take Pick from the center
```

```
 location and place at the final destination. All Arms can
 reach the center location."
|point|

point<-Place getGrip.

Arm<-''.
STATE<-#state8.
ERROR<-0.
TaskPlane
   associationsDo:[:each|(each value isArm)
                        ifTrue:[(each value canReach:point)
                              ifTrue:[(Arm == '')
                           ifTrue:[Arm<-each key] ] ] ].

(Arm == '') ifTrue:[ERROR<-1].

state8
"Have the second arm approach the center location"
|point|

point<-(TaskPlane at:#center) getGrip.

ERROR<-(TaskPlane at:Arm) approach:point.
STATE<-#state9.

state9
"Have Arm grip Pick"
ERROR<-(TaskPlane at:Arm) grip:Pick.
STATE<-#state10.

state10
"Depart the Arm while carrying Pick"

ERROR<-(TaskPlane at:Arm) depart.
STATE<-#state11.

state11
"Have Arm approach final destination while carrying Pick"
|point|

point<-Place getGrip.
ERROR<-(TaskPlane at:Arm) approach:point.
STATE<-#state12.
```

```
state12
"Release Pick at final destination (Place)"
ERROR<-(TaskPlane at:Arm) release.
STATE<-#home.

home
"Release object in arm and return arm home"
(TaskPlane at:Arm) release.
(TaskPlane at:Arm) home.
ERROR<-0.
STATE<-#exit.

exit
"return all arms to their home positions and set Exit to true."
TaskPlane
   associationsDo:[ :each|
                (each value isArm) ifTrue:[each value release]].
Exit<-true.



!Task methodsFor: 'commands'!

pick:item place:loc
"initialize state table and go through the states
 until Exit becomes true."

Pick<- (item asLowercase) asSymbol.
Place<- (loc asLowercase) asSymbol.
self initView.
self initTable.
self initPlane.
[Exit]
   whileFalse:[
            self perform:(stateTable at:
              (((STATE printString), (ERROR printString)) asSymbol)) ].
^'bye'

Task class
        instanceVariableNames: ''!


!Task class methodsFor: 'instance creation'!
```

```
pick:item place:loc
|tempTask|
tempTask<-super new.
^tempTask pick:item place:loc


Task subclass: #TaskArm1
        instanceVariableNames: 'anObject arm1Rec home center gripPoint
          reachX reachY '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!

!TaskArm1 methodsFor: 'initialization'!

setValues
arm1Rec<-Quadrangle new.
arm1Rec insideColor:Form black.
arm1Rec extent:20@20.
arm1Rec<-arm1Rec translateBy:(AView origin).
arm1Rec<- arm1Rec translateBy:(100@80).
center<-arm1Rec center.
home<-arm1Rec origin.
anObject<-''.
gripPoint<-center.
reachX<-200.
reachY<-200.
self addDependent:AView.


!TaskArm1 methodsFor: 'get values'!

getGrip
^gripPoint

getMaxX
^(arm1Rec topRight) x

getMaxY
^(arm1Rec bottomRight) y

getName
^'arm1'
```

```
!TaskArm1 methodsFor: 'queries'!

canReach:point
"point is the center point of some object. "
|x y|

x<-((arm1Rec center) x) + reachX.
y<-((arm1Rec center) y) + reachY.
^((x >= object x) & (y >= object y))

isArm
^true

mounted
^true


!TaskArm1 methodsFor: 'perform tasks'!

approach:point
"return 0 for success and 1 for error"

arm1Rec moveTo:point.
(anObject ~= '') ifTrue:[anObject approach:point].
self changed.
^0

depart
"move up three places on the y axis. If carrying something,
 send message to move it also"

arm1Rec moveTo:(((arm1Rec corner) x)@(((arm1Rec corner) y) -3)).
(anObject ~= '') ifTrue:[anObject depart].
^0

display
"display the arm on the display view"
arm1Rec display.
^0

grip:object
"object is an instance of some object"

arm1Rec moveTo:(object getGrip).
anObject<-object.
```

```
^0

home
"move to its original location"

arm1Rec moveTo:home.
self changed.
^0

release
"release the object the arm is carrying"

(anObject == '') ifTrue:[anObject release. anObject<-''].
^0
```

```
TaskArm1 class
        instanceVariableNames: ''!


!TaskArm1 class methodsFor: 'instance creation'!

new
|tempArm|

tempArm <- super new.
tempArm setValues.
^tempArm

Task subclass: #TaskArm2
        instanceVariableNames: 'anObject arm2Rec home center gripPoint
            reachX reachY '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!


!TaskArm2 methodsFor: 'initialization'!

setValues

arm2Rec<-Quadrangle new.
arm2Rec insideColor:Form black.
arm2Rec extent:20@20.
arm2Rec<-arm2Rec translateBy:(AView origin).
```

```
arm2Rec<- arm2Rec translateBy:(200@80).
center<-arm2Rec center.
home<-arm2Rec origin.
anObject<-''.
gripPoint<-center.
reachX<-50.
reachY<-50.
self addDependent:AView.


!TaskArm2 methodsFor: 'get values'!

getGrip
^arm2Rec center

getMaxX
^(arm2Rec topRight) x

getMaxY
^(arm2Rec bottomRight) y

getName
^'arm2'!


!TaskArm2 methodsFor: 'queries'!

canReach:point
"point is the center point of some object. "
|x y|

x<-((arm2Rec center) x) + reachX.
y<-((arm2Rec center) y) + reachY.
^((x >= object x) & (y >= object y))

isArm
^true

mounted
^true


!TaskArm2 methodsFor: 'perform tasks'!

approach:point
```

```
"return 0 for success and 1 for error"

arm2Rec moveTo:point.
(anObject ~= '') ifTrue:[anObject approach:point].
self changed.
^0

depart
"move up three places on the y axis. If carrying something, send
 message to move it also"

arm2Rec moveTo:(((arm2Rec corner) x)@(((arm2Rec corner) y) -3)).
(anObject ~= '') ifTrue:[anObject depart].
self changed.
^0

display
"display the arm on the display view"
arm2Rec display.
^0

grip:object
"object is an instance of some object"

arm2Rec moveTo:(object getGrip).
anObject<-object.
^0

home
"move to its original location"

arm2Rec moveTo:home.
self changed.
^0

release
"release the object the arm is carrying"
(anObject == '') ifTrue:[anObject release. anObject<-''].
^0

TaskArm2 class
        instanceVariableNames: '!
```

```
!TaskArm2 class methodsFor: 'instance creation'!

new
|tempArm|

tempArm <- super new.
tempArm setValues.
^tempArm

Task subclass: #TaskBook
        instanceVariableNames: 'anObject bookRec home center gripPoint '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!


!TaskBook methodsFor: 'initialization'!

setValues

bookRec<-Quadrangle new.
bookRec insideColor:Form black.
bookRec extent:20@20.
bookRec<-bookRec translateBy:(AView origin).
bookRec<- bookRec translateBy:(50@80).
center<-bookRec center.
home<-bookRec origin.
anObject<-''.
gripPoint<-center.
self addDependent:AView.


!TaskBook methodsFor: 'get values'!

getGrip
^gripPoint

getMaxX
^(bookRec topRight) x

getMaxY
^(bookRec bottomRight) y

getName
^'book'!
```

!TaskBook methodsFor: 'queries'!

canReach:point
"this only has meaning for an arm. Return 1 for failure"
^1

isArm
^false

mounted
^false

!TaskBook methodsFor: 'perform tasks'!

approach:point
bookRec moveTo:point.
self changed.
^0

depart
"move up three places"

bookRec moveTo:(((bookRec corner) x)@(((bookRec corner) y) -3)).
self changed.
^0

display
"display the book on the display view"
bookRec display.
^0

grip:object
"this only has meaning for an arm"
^1

home
"move to its original location"

bookRec moveTo:home.
self changed.
^0

release

```
"only has meaning for the an arm"
^0


TaskBook class
        instanceVariableNames: ''!


!TaskBook class methodsFor: 'instance creation'!

new
|tempObj|
tempObj <- super new.
tempObj setValues.
^tempObj

Task subclass: #TaskBox
        instanceVariableNames: 'anObject boxRec home center gripPoint '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!


!TaskBox methodsFor: 'initialization'!

setValues

boxRec<-Quadrangle new.
boxRec insideColor:Form black.
boxRec extent:20@20.
boxRec<-boxRec translateBy:(AView origin).
boxRec<- boxRec translateBy:(250@80).
center<-boxRec center.
home<-boxRec origin.
anObject<-''.
gripPoint<-center.
self addDependent:AView.

!TaskBox methodsFor: 'get values'!

getGrip
^gripPoint
```

```
getMaxX
^(boxRec topRight) x

getMaxY
^(boxRec bottomRight) y

getName
^'box'!


!TaskBox methodsFor: 'queries'!

canReach:point
"this only has meaning for an arm. Return 1 for failure"
^1

isArm
^false

mounted
^false


!TaskBox methodsFor: 'perform tasks'!

approach:point
boxRec moveTo:point.
self changed.
^0

depart
"move up three places"

boxRec moveTo:(((boxRec corner) x)@(((boxRec corner) y) -3)).
self changed.
^0

display
"display the box on the display view"
boxRec display.
^0

grip:object
"this only has meaning for an arm"
^1
```

```
home
"move to its original location"

boxRec moveTo:home.
self changed.
↑0

release
"only has meaning for the an arm"
↑0

TaskBox class
        instanceVariableNames: "!

!TaskBox class methodsFor: 'instance creation'!

new
|tempObj|

tempObj <- super new.
tempObj setValues.
^tempObj

Task subclass: #TaskCenter
        instanceVariableNames: 'centerRec home centerPt '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!


!TaskCenter methodsFor: 'initialization'!

setValues

centerRec<-Quadrangle new.
centerRec insideColor:Form black.
centerRec extent:1@1.
centerRec<-centerRec translateBy:(AView origin).
centerRec<- centerRec translateBy:(150@80).
centerPt<-centerRec center.
home<-centerRec origin.
self addDependent:AView.
```

```
!TaskCenter methodsFor: 'get values'!

getGrip
^centerRec center

getMaxX
^(centerRec topRight) x

getMaxY
^(centerRec bottomRight) y

getName
^'center'!

!TaskCenter methodsFor: 'queries'!

canReach:point
"the center of the plane can not reach anything"
^false

isArm
^false

mounted
^true


!TaskCenter methodsFor: 'perform tasks'!

approach:point
^1

depart
^1

display
"display the center as a point on the display view"
centerRec display.
^0

grip:object
^1
```

```
home
^0

release
^1

TaskCenter class
        instanceVariableNames: ''!


!TaskCenter class methodsFor: 'instance creation'!

new
|tempCntr|

tempCntr <- super new.
tempCntr setValues.
^tempCntr

StandardSystemView subclass: #TaskView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix-II'!


!TaskView methodsFor: 'accessing'!

origin
^self viewport origin


!TaskView methodsFor: 'displaying'!

displayView
"send message display to all instances within TaskPlane"

super displayView
model size > 0
  ifTrue:[
        TaskPlane associationsDo:[ :each|
            each value display]].

update
"the objects have moved around"
```

```
self display.

update:aModel
self display.
```

VITA

Torez Hiley, the daughter of Grethel and Karriem Bey, was born on October 5, 1959 in Chicago, Illinois. She attended University of Illinois and received a Bachelor of Science Degree in Math with a minor in Computer Science, June of 1982. In January of 1983, she began her career with AT&T as a System Analyst. In February of 1985 she was transferred to Summit, New Jersey where she continued her work as a development engineer on the Unix Operating System. In January, 1985 she began her Master of Science Degree studies at Lehigh University. She majored in Computer Engineering and received her degree in December 1987.